CakePHP Cookbook Documentation

Version 2.x

Cake Software Foundation

30 July 2013

Table des matières

1 Pour Commencer **1**
Tutoriel d'un Blog . . . 1
Blog Tutoriel - Ajouter la logique . . . 9

2 Installation **29**
Conditions requises . . . 29
Licence . . . 29
Télécharger CakePHP . . . 30
Permissions . . . 30
Configuration . . . 30
Développement . . . 30
Production . . . 31
Installation avancée et URL Rewriting . . . 32
A vous de jouer ! . . . 39

3 Débuter avec CakePHP **41**
Qu'est ce que CakePHP ? Pourquoi l'utiliser ? . . . 41
Comprendre le système M-V-C (Model-View-Controller) . . . 42
Où obtenir de l'aide . . . 44

4 Controllers (Contrôleurs) **47**
Le Controller App . . . 47
Les paramètres de requête . . . 48
Les Actions du Controller . . . 48
Request Life-cycle callbacks . . . 50
Les Méthodes du Controller . . . 50
Les attributs du Controller . . . 58
En savoir plus sur les controllers . . . 59

5 Views (Vues) **81**

Templates de Views . . . 81
Utiliser les blocs de vues . . . 83
Layouts . . . 85
Elements . . . 88
Créer vos propres classes de vue . . . 91
API de View . . . 91
En savoir plus sur les vues . . . 93

6 Models (Modèles) 105
Comprendre les Models . . . 105
Pour en savoir plus sur les models . . . 107

7 Librairies du Coeur 209
Usage Général . . . 209
Behaviors (Comportements) . . . 248
Components (Composants) . . . 273
Helpers (Assistants) . . . 325
Utilitaires . . . 413

8 Plugins 525
Installer un Plugin . . . 525
Configuration du Plugin . . . 526
Aller plus loin avec le bootstrapping . . . 526
Utiliser un Plugin . . . 527
Créer Vos Propres Plugins . . . 527
Controllers du Plugin . . . 528
Models du Plugin . . . 529
Vues du Plugin . . . 530
Plugin assets . . . 530
Components, Helpers et Behaviors . . . 531
Etendez votre Plugin . . . 532
Astuces pour les Plugins . . . 532

9 Console et Shells 533
La console de CakePHP . . . 533
Créer un shell . . . 535
Les tâches Shell . . . 537
Invoquer d'autres shells à partir de votre shell . . . 538
Niveaux de sortie de la Console . . . 538
Style de sortie . . . 539
Configurer les options et générer de l'aide . . . 540
Routing dans shells / CLI . . . 547
API de Shell . . . 548
Plus de sujets . . . 551

10 Développement 565
Configuration . . . 565
Routing . . . 578

Sessions . . . 595
Exceptions . . . 601
Gestion des Erreurs . . . 608
Debugger . . . 611
Testing . . . 614
REST . . . 637
Filtres du Dispatcher . . . 641
Les packages Vendor . . . 645

11 Déploiement **647**
Vérifier votre sécurité . . . 647
Définir le document root . . . 647
Mise à jour de core.php . . . 647
Améliorer les performances de votre application . . . 648

12 Tutoriels et exemples **649**
Tutoriel d'un Blog . . . 649
Blog Tutoriel - Ajouter la logique . . . 657
Authentification Simple et Autorisation de l'Application . . . 668
Application Simple contrôlée par Acl . . . 675
Application Simple contrôlée par Acl - partie 2 . . . 682

13 Annexes **687**
2.4 Guide de Migration . . . 687
2.3 Guide de Migration . . . 692
2.2 Guide de Migration . . . 696
2.1 Guide de Migration . . . 702
2.0 Guide de Migration . . . 713
Migration de la version 1.2 vers la 1.3 . . . 745
Informations générales . . . 763

14 Indices et tables **767**

Index **769**

CHAPITRE 1

Pour Commencer

Le framework CakePHP fournit une base robuste pour votre application. Il peut gérer tous les aspects, de la requête initiale de l'utilisateur et son cheminement jusqu'au rendu final de la page web. Et puisque le framework suit les principes du MVC, il vous permet de facilement personnaliser et offre la plupart des aspects de votre application.

Le framework fournit aussi une structure organisationnelle basique, des noms de fichier jusqu'aux noms des tables de la base de données, en gardant toute votre application cohérente et logique. Ce concept est simple mais puissant. Suivez les conventions et vous saurez toujours exactement où les choses se trouvent et comment elles sont organisées.

La meilleure façon de découvrir et d'apprendre CakePHP est de s'assoir et de construire quelque chose. Pour commencer, nous construirons une application simple de blog.

Tutoriel d'un Blog

Bienvenue sur CakePHP. Vous consultez probablement ce tutoriel parce que vous voulez en apprendre plus à propos du fonctionnement de CakePHP. C'est notre but d'améliorer la productivité et de rendre le développement plus agréable : nous espérons que vous le découvrirez au fur et à mesure que vous plongerez dans le code.

Ce tutoriel vous accompagnera à travers la création d'une simple application de blog. Nous récupérerons et installerons Cake, créerons et configurerons une base de données et ajouterons suffisamment de logique applicative pour lister, ajouter, éditer et supprimer des posts.

Voici ce dont vous aurez besoin :

1. Un serveur web fonctionnel. Nous supposerons que vous utilisez Apache, bien que les instructions pour utiliser d'autres serveurs devraient être très semblables. Nous aurons peut-être besoin de jouer un peu sur la configuration du serveur, mais la plupart des personnes peuvent faire fonctionner Cake sans aucune configuration préalable.
2. Un serveur de base de données. Dans ce tutoriel, nous utiliserons MySQL. Vous aurez besoin d'un minimum de connaissance en SQL afin de créer une base de données : Cake prendra les rênes à partir de là.

3. Des connaissances de base en PHP. Plus vous aurez d'expérience en programmation orienté objet, mieux ce sera ; mais n'ayez crainte, même si vous êtes adepte de la programmation procédurale.
4. Enfin, vous aurez besoin de connaissances de base à propos du motif de conception MVC. Un bref aperçu de ce motif dans le chapitre *Comprendre le système M-V-C (Model-View-Controller)*. Ne vous inquiétez pas : il n'y a qu'une demi-page de lecture.

Maintenant, lançons-nous !

Obtenir Cake

Tout d'abord, récupérons une copie récente de Cake.

Pour obtenir la dernière version, allez sur le site GitHub du projet CakePHP : https ://github.com/cakephp/cakephp/tags et téléchargez la dernière version de la 2.0.

Vous pouvez aussi cloner le dépôt en utilisant git (http ://git-scm.com/). `git clone git://github.com/cakephp/cakephp.git`

Peu importe comment vous l'avez téléchargé, placez le code à l'intérieur du "DocumentRoot" de votre serveur. Une fois terminé, votre répertoire d'installation devrait ressembler à quelque chose comme cela :

```
/chemin_du_document_root
    /app
    /lib
    /plugins
    /vendors
    .htaccess
    index.php
    README
```

A présent, il est peut-être temps de voir un peu comment fonctionne la structure de fichiers de Cake : lisez le chapitre *Structure du dossier de CakePHP*.

Créer la base de données du blog

Maintenant, mettons en place la base de données pour notre blog. Si vous ne l'avez pas déjà fait, créez une base de données vide avec le nom de votre choix pour l'utiliser dans ce tutoriel. Pour le moment, nous allons juste créer une simple table pour stocker nos posts. Nous allons également insérer quelques posts à des fins de tests. Exécutez les requêtes SQL suivantes dans votre base de données :

```
/* D'abord, créons la table des posts : */
CREATE TABLE posts (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);

/* Puis insérons quelques posts pour les tests : */
INSERT INTO posts (title,body,created)
```

```
    VALUES ('Le titre', 'Voici le contenu du post.', NOW());
INSERT INTO posts (title,body,created)
    VALUES ('Encore un titre', 'Et le contenu du post qui suit.', NOW());
INSERT INTO posts (title,body,created)
    VALUES ('Le retour du titre', 'C\'est très excitant, non ?', NOW());
```

Le choix des noms pour les tables et les colonnes ne sont pas arbitraires. Si vous respectez les conventions de nommage Cake pour les bases de données et les classes (toutes deux expliquées le chapitre *Conventions de CakePHP*), vous tirerez profit d'un grand nombre de fonctionnalités automatiques et vous éviterez des étapes de configurations. Cake est suffisamment souple pour implémenter les pires schémas de bases de données, mais respecter les conventions vous fera gagner du temps.

Consultez le chapitre *Conventions de CakePHP* pour plus d'informations, mais il suffit de comprendre que nommer notre table 'posts' permet de la relier automatiquement à notre model Post, et qu'avoir des champs 'modified' et 'created' permet de les avoir gérés automagiquement par Cake.

Configurer la base de données Cake

En avant : indiquons à Cake où se trouve notre base de données et comment s'y connecter. Pour la plupart d'entre vous, c'est la première et dernière fois que vous configurerez quelque chose.

Une copie du fichier de configuration Cake pour la base de données se trouve dans `/app/Config/database.php.default`. Faites une copie de ce fichier dans le même répertoire mais nommez le `database.php`.

Le fichier de configuration devrait être assez simple : remplacez simplement les valeurs du tableau `$default` par celles qui correspondent à votre installation. Un exemple de tableau de configuration complet pourrait ressembler à ce qui suit :

```
public $default = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'port' => '',
    'login' => 'cakeBlog',
    'password' => 'c4k3-rUl3Z',
    'database' => 'cake_blog_tutorial',
    'schema' => '',
    'prefix' => '',
    'encoding' => ''
);
```

Une fois votre nouveau fichier `database.php` sauvegardé, vous devriez être en mesure d'ouvrir votre navigateur internet et de voir la page d'accueil de Cake. Elle devrait également vous indiquer votre fichier de connexion a été trouvé, et que Cake peut s'y connecter avec succès.

Note : Rappelez-vous que vous aurez besoin d'avoir PDO, et pdo_mysql activés dans votre php.ini.

Configuration facultative

Il y a trois autres élements qui peuvent être configurés. La plupart des développeurs configurent les éléments de cette petite liste, mais ils ne sont pas obligatoires pour ce tutoriel. Le premier consiste à définir une chaîne de caractères personnalisée (ou "grain de sel") afin de sécuriser les hashs. Le second consiste à définir un nombre personnalisé (ou "graine") à utiliser pour le chiffrage. Le troisième est de permettre l'accès en écriture à CakePHP pour son dossier `tmp`.

Le "grain de sel" est utilisé pour générer des hashes. Changez sa valeur par défaut en modifiant `/app/Config/core.php` à la ligne 187. La nouvelle valeur n'a pas beaucoup d'importance du moment qu'elle est difficile à deviner :

```
/**
 * Une chaîne aléatoire utilisée dans les méthodes de hachage sécurisées.
 */
Configure::write('Security.salt', 'pl345e-P45s_7h3*S@l7!');
```

La "graine cipher" est utilisée pour le chiffrage/déchiffrage des chaînes de caractères. Changez sa valeur par défaut en modifiant `/app/Config/core.php` à la ligne 192. La nouvelle valeur n'a pas beaucoup d'importance du moment qu'elle est difficile à deviner :

```
/**
 * Une chaîne aléatoire de chiffre utilisée pour le chiffrage/déchiffrage
 * des chaînes de caractères.
 */
Configure::write('Security.cipherSeed', '7485712659625147843639846751');
```

La dernière étape consiste à rendre le dossier `/app/tmp` accessible en écriture. Le meilleur moyen de faire cela est de trouver sous quel utilisateur votre serveur web s'exécute (`<?php echo `whoami`; ?>`) et de modifier le propriétaire du dossier `/app/tmp` pour cet utilisateur. La commande à exécuter (sous *nix) devrait ressembler à quelque chose comme cela

```
$ chown -R www-data app/tmp
```

Si pour une raison quelconque CakePHP ne peut pas écrire dans ce répertoire, vous en serez informé par un message d'avertissement tant que vous n'êtes pas en mode production.

Une note sur mod_rewrite

Occasionnellement, un nouvel utilisateur peut avoir des problèmes de mod_rewrite. par exemple si la page d'accueil de CakePHP a l'air marrante (pas d'images ou de styles css), cela signifie probablement que mod_rewrite ne fonctionne pas sur votre système. Merci de vous référer à la section suivante sur l'url rewriting pour que votre serveur web fonctionne :

URL Rewriting

Apache et mod_rewrite (et .htaccess)

Alors que CakePHP est construit pour travailler avec mod_rewrite –et habituellement il l'est– nous avons remarqué que certains utilisateurs se battent pour obtenir un bon fonctionnement sur leurs systèmes.

Ici il y a quelques trucs que vous pourriez essayer pour que cela fonctionne correctement. Premièrement, regardez votre fichier httpd.conf (Assurez vous que vous avez éditer le httpd.conf du système plutôt que celui d'un utilisateur- ou le httpd.conf d'un site spécifique).

1. Assurez-vous qu'un .htaccess est permis et que AllowOverride est défini à All pour le DocumentRoot correct. Vous devriez voir quelque chose comme :

```
# Chaque répertoire auquel Apache a accès peut être configuré avec
# respect pour lesquels les services et les fonctionnalités sont
# autorisés et/ou désactivés dans ce répertoire (et ses sous-répertoires).
#
# Premièrement, nous configurons "par défault" pour être un ensemble
# très restrictif de fonctionnalités.
#
<Directory />
    Options FollowSymLinks
    AllowOverride All
#    Order deny,allow
#    Deny from all
</Directory>
```

2. Assurez-vous que vous avez chargé correctement mod_rewrite. Vous devriez voir quelque chose comme :

```
LoadModule rewrite_module libexec/apache2/mod_rewrite.so
```

   Dans la plupart des systèmes, ceux-ci vont être commentés (en étant précédé par un #) par défaut, donc vous aurez juste besoin de retirer les symboles # du début.

   Après que vous avez fait des changements, re-démarrez Apache pour être sûr que les paramètres soient actifs.

   Vérifiez que vos fichiers .htaccess sont effectivement dans le bon répertoire.

   Cela peut arriver pendant la copie parce que certains systèmes d'exploitation traitent les fichiers qui commencent par '.' en caché et du coup ne les voient pas pour les copier.

3. Assurez-vous que votre copie de CakePHP vient de la section des téléchargements du site de notre dépôt GIT, et a été dézippé correctement en vérifiant les fichiers .htaccess.

   Le répertoire root de Cake (a besoin d'être copié dans votre document, cela redirige tout vers votre app Cake) :

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$ app/webroot/    [L]
   RewriteRule    (.*) app/webroot/$1 [L]
</IfModule>
```

   Le répertoire app de Cake (sera copié dans le répertoire supérieur de votre application avec Bake) :

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$     webroot/    [L]
   RewriteRule    (.*) webroot/$1    [L]
</IfModule>
```

Le répertoire webroot de Cake (sera copié dans le webroot de votre application avec Bake) :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php/$1 [QSA,L]
</IfModule>
```

Si votre site Cakephp a toujours des problèmes avec mod_rewrite, essayez de modifier les paramètres pour les virtualhosts. Si vous êtes sur ubuntu, modifiez le fichier /etc/apache2/sites-available/default (l'endroit dépend de la distribution). Dans ce fichier, assurez-vous que `AllowOverride None` a changé en `AllowOverride All`, donc vous avez :

```
<Directory />
    Options FollowSymLinks
    AllowOverride All
</Directory>
<Directory /var/www>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order Allow,Deny
    Allow from all
</Directory>
```

Si vous êtes sur Mac OSX, une autre solution est d'utiliser l'outil virtualhostx pour faire un hôte virtuel pour pointer vers votre dossier.

Pour beaucoup de services d'hébergement (GoDaddy, 1and1), votre serveur web est en fait déjà distribué à partir d'un répertoire utilisateur qui utilise déjà mod_rewrite. Si vous installez CakePHP dans un répertoire utilisateur (http ://exemple.com/~username/cakephp/), ou toute autre structure d'URL qui utilise déjà mod_rewrite, vous aurez besoin d'ajouter les requêtes (statements) RewriteBase aux fichiers .htaccess que CakePHP utilise (/.htaccess, /app/.htaccess, /app/webroot/.htaccess).

Ceci peut être ajouté à la même section avec la directive RewriteEngine, donc par exemple, votre fichier .htaccess dans webroot ressemblerait à ceci :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/cake/app
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php/$1 [QSA,L]
</IfModule>
```

Les détails de ces changements dépendront de votre configuration, et pourront inclure des choses supplémentaires qui ne sont pas liées à CakePHP. Merci de vous renseigner sur la documentation en ligne d'Apache pour plus d'informations.

4. (Optionel) Pour améliorer la configuration de production, vous devriez empêcher les assets invalides d'être parsés par CakePHP. Modifier votre webroot .htaccess pour quelque chose comme :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/cake/app
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_URI} !^/(app/webroot/)?(img|css|js)/(.*)$
    RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

Ce qui est au-dessus va simplement empêcher les assets incorrects d'être envoyés à index.php et à la place d'afficher la page 404 de votre serveur web.

De plus, vous pouvez créer une page HTML 404 correspondante, ou utiliser la page 404 de CakePHP intégrée en ajoutant une directive `ErrorDocument` :

```
ErrorDocument 404 /404-not-found
```

De belles URLs sur nginx

nginx est un serveur populaire qui, comme Lighttpd, utilise moins de ressources système. Son inconvénient est qu'il ne fait pas usage de fichiers .htaccess comme Apache et Lighttpd, il est donc nécessaire de créer les URLs réécrites dans la configuration du site disponibles. selon votre configuration, vous devrez modifier cela, mais à tout le moins, vous aurez besoin de PHP fonctionnant comme une instance FastCGI.

```
server {
    listen   80;
    server_name www.example.com;
    rewrite ^(.*) http://example.com$1 permanent;
}

server {
    listen   80;
    server_name example.com;

    # root directive should be global
    root   /var/www/example.com/public/app/webroot/;
    index  index.php;

    access_log /var/www/example.com/log/access.log;
    error_log /var/www/example.com/log/error.log;

    location / {
        try_files $uri $uri/ /index.php?$uri&$args;
    }

    location ~ \.php$ {
        include /etc/nginx/fastcgi_params;
        try_files $uri =404;
        fastcgi_pass    127.0.0.1:9000;
```

```
        fastcgi_index   index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

Rewrites d'URL sur IIS7 (serveurs Windows)

IIS7 ne supporte pas nativement les fichiers .htaccess. Bien qu'il existe des add-ons qui peuvent ajouter ce support, vous pouvez aussi importer les règles des .htaccess dans IIS pour utiliser les rewrites natifs de CakePHP. Pour ce faire, suivez ces étapes :

1. Utilisez l'installeur de la plateforme Web de Microsoft (http ://www.microsoft.com/web/downloads/platform.aspx) pour installer l'URL Rewrite Module 2.0 (http ://www.iis.net/downloads/microsoft/url-rewrite) ou télécharger le directement (32-bit (http ://www.microsoft.com/en-us/download/details.aspx ?id=5747) / 64-bit (http ://www.microsoft.com/en-us/download/details.aspx ?id=7435)).
2. Créez un nouveau fichier dans votre dossier CakePHP, appelé web.config.
3. Utilisez Notepad ou tout autre éditeur XML-safe, copiez le code suivant dans votre nouveau fichier web.config...

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="Rewrite requests to test.php" stopProcessing="true">
                    <match url="^test.php(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="app/webroot/test.php{R:1}" />
                </rule>
                <rule name="Exclude direct access to app/webroot/*" stopProcessing="true">
                    <match url="^app/webroot/(.*)$" ignoreCase="false" />
                    <action type="None" />
                </rule>
                <rule name="Rewrite routed access to assets (img, css, files, js, favicon)"
                    <match url="^(img|css|files|js|favicon.ico)(.*)$" />
                    <action type="Rewrite" url="app/webroot/{R:1}{R:2}" appendQueryString="
                </rule>
                <rule name="Rewrite requested file/folder to index.php" stopProcessing="tru
                    <match url="^(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="index.php" appendQueryString="true" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

Une fois que le fichier web.config est créé avec les bonnes règles de réécriture des liens de IIS, les liens CakePHP, les CSS, les JS, et le reroutage devraient fonctionner correctement.

Je ne veux / ne peux utiliser l'URL rewriting

Si vous ne voulez ou ne pouvez pas utiliser l'URL rewriting sur votre serveur web, référez-vous à la section *core configuration*.

Maintenant continuez sur *Blog Tutoriel - Ajouter la logique* pour commencer à construire votre première application CakePHP.

Blog Tutoriel - Ajouter la logique

Créer un model Post

La classe Model est le pain quotidien des applications CakePHP. En créant un model CakePHP qui interagira avec notre base de données, nous aurons mis en place les fondations nécessaires pour faire plus tard nos opérations de lecture, d'insertion, d'édition et de suppression.

Les fichiers des classes Model de CakePHP se trouvent dans `/app/Model`, et le fichier que nous allons créer maintenant sera enregistré dans `/app/Model/Post.php`. Le fichier complet devrait ressembler à ceci

```
class Post extends AppModel {
}
```

La convention de nommage est vraiment très importante dans CakePHP. En nommant notre model Post, CakePHP peut automatiquement déduire que ce model sera utilisé dans le controller PostsController, et sera lié à la table `posts` de la base de données.

Note : CakePHP créera dynamiquement un objet model pour vous, s'il ne trouve pas le fichier correspondant dans /app/Model. Cela veut aussi dire que si vous n'avez pas nommé correctement votre fichier (par ex. post.php ou posts.php). CakePHP ne reconnaîtra pas votre configuration et utilisera ses objets model par défaut.

Pour plus d'informations sur les models, comme les préfixes des tables, les callbacks, et la validation, consultez le chapitre *Models (Modèles)* du manuel.

Créer un controller Posts

Nous allons maintenant créer un controller pour nos posts. Le controller est l'endroit où s'exécutera toute la logique métier pour l'intéraction du processus de post. En un mot, c'est l'endroit où vous jouerez avec les models et où les tâches liées aux posts s'exécutent. Nous placerons ce nouveau controller dans un fichier appelé `PostsController.php` à l'intérieur du dossier `/app/Controller`. Voici à quoi devrait ressembler le controller de base

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');
}
```

Maintenant, ajoutons une action à notre controller. Les actions représentent souvent une simple fonction ou une interface dans une application. Par exemple, quand les utilisateurs requêtent www.exemple.com/posts/index (qui est également la même chose que www.exemple.com/posts/), ils pourraient s'attendre à voir une liste de posts. Le code pour cette action devrait ressembler à quelque chose comme ça :

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');

    public function index() {
        $this->set('posts', $this->Post->find('all'));
    }
}
```

Laissez-moi vous expliquer un peu cette action. En définissant la fonction `index()` dans notre PostsController, les utilisateurs peuvent maintenant accéder à cette logique en demandant www.exemple.com/posts/index. De la même façon, si nous devions définir une fonction appelée `foobar()`, les utilisateurs pourrait y accéder en demandant www.exemple.com/posts/foobar.

Warning : Vous pourriez être tenté de nommer vos controllers et vos actions d'une certaine manière pour obtenir une certaine URL. Résistez à cette tentation. Suivez les conventions CakePHP (le nom des controllers au pluriel, etc.) et nommez vos actions de façon lisible et compréhensible. Vous pouvez lier les URLs à votre code en utilisant ce qu'on appelle des "routes", on le verra plus tard.

La seule instruction que cette action utilise est `set()`, pour transmettre les données du controller à la vue (que nous créerons à la prochaine étape). La ligne définit la variable de vue appelée 'posts' qui est égale à la valeur de retour de la méthode `find('all')` du model Post. Notre model Post est automatiquement disponible via $this->Post, parce que nous avons suivi les conventions de nommage de Cake.

Pour en apprendre plus sur les controllers de Cake, consultez le chapitre *Controllers (Contrôleurs)*.

Créer les Vues des Posts

Maintenant que nous avons nos données en provenance du model, ainsi que la logique applicative et les flux définis par notre controller, nous allons créer une vue pour l'action "index" que nous avons créé ci-dessus.

Les vues de Cake sont juste des fragments de présentation "assaisonnée", qui s'intègrent au sein d'un layout applicatif. Pour la plupart des applications, elles sont un mélange de HTML et PHP, mais les vues peuvent aussi être constituées de XML, CSV ou même de données binaires.

Les Layouts sont du code de présentation, encapsulé autour d'une vue, ils peuvent être définis et interchangés, mais pour le moment, utilisons juste celui par défaut.

Vous souvenez-vous, dans la dernière section, comment nous avions assigné la variable 'posts' à la vue en utilisant la méthode `set()` ? Cela devrait transmettre les données à la vue qui ressemblerait à quelque chose comme ça :

```
// print_r($posts) sort:

Array
```

```
(
    [0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => Le titre
                    [body] => Voici le contenu du post.
                    [created] => 2008-02-13 18:34:55
                    [modified] =>
                )
        )
    [1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => Encore un titre
                    [body] => Et le contenu du post qui suit.
                    [created] => 2008-02-13 18:34:56
                    [modified] =>
                )
        )
    [2] => Array
        (
            [Post] => Array
                (
                    [id] => 3
                    [title] => Le retour du titre
                    [body] => C'est très excitant, non ?
                    [created] => 2008-02-13 18:34:57
                    [modified] =>
                )
        )
)
```

Les fichiers des vues de Cake sont stockés dans `/app/views` à l'intérieur d'un dossier dont le nom correspond à celui du controller (nous aurons à créer un dossier appelé 'posts' dans ce cas). Pour mettre en forme les données de ces posts dans un joli tableau, le code de notre vue devrait ressembler à quelque chose comme cela :

```
<!-- File: /app/View/Posts/index.ctp -->

<h1>Blog posts</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
    </tr>

    <!-- Here is where we loop through our $posts array, printing out post info -->
```

```
    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'],
            array('controller' => 'posts', 'action' => 'view', $post['Post']['id'])); ?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
    <?php endforeach; ?>
    <?php unset($post); ?>
</table>
```

Bien entendu, cela donnera quelque chose de simple.

Vous avez sans doute remarqué l'utilisation d'un objet appelé `$this->Html`. C'est une instance de la classe CakePHP `HtmlHelper`. CakePHP est livré avec un ensemble de "helpers" (des assistants) pour les vues, qui réalisent en un clin d'oeil des choses comme le "linking" (mettre les liens dans un texte), l'affichage des formulaires, du JavaScript et de l'Ajax. Vous pouvez en apprendre plus sur la manière de les utiliser dans le chapitre *Helpers (Assistants)*, mais ce qu'il est important de noter ici, c'est que la méthode `link()` génèrera un lien HTML à partir d'un titre (le premier paramètre) et d'une URL (le second paramètre).

Lorsque vous indiquez des URLs dans Cake, il est recommandé d'utiliser les tableaux. Ceci est expliqué dans le chapitre des Routes. Utilisez les tableaux dans les URLs, vous permet de tirer profit des capacités de CakePHP à ré-inverser les routes. Vous pouvez aussi utiliser les URLs relatives depuis la base de l'application comme suit /controller/action/param1/param2.

A ce stade, vous devriez être en mesure de pointer votre navigateur sur la page http ://www.exemple.com/posts/index. Vous devriez voir votre vue, correctement formatée avec le titre et le tableau listant les posts.

Si vous avez essayé de cliquer sur l'un des liens que nous avons créés dans cette vue (le lien sur le titre d'un post mène à l'URL : /posts/view/un_id_quelconque), vous avez sûrement été informé par CakePHP que l'action n'a pas encore été définie. Si vous n'avez pas été informé, soit quelque chose s'est mal passé, soit en fait vous aviez déjà défini l'action, auquel cas vous êtes vraiment sournois ! Sinon, nous allons la créer sans plus tarder dans le Controller Posts

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');

    public function index() {
         $this->set('posts', $this->Post->find('all'));
    }

    public function view($id = null) {
        if (!$id) {
            throw new NotFoundException(__('Invalid post'));
        }

        $post = $this->Post->findById($id);
        if (!$post) {
            throw new NotFoundException(__('Invalid post'));
        }
```

```
        $this->set('post', $post);
    }
}
```

L'appel de `set()` devrait vous être familier. Notez que nous utilisons `read()` plutôt que `find('all')` parce que nous voulons seulement récupérer les informations d'un seul post.

Notez que notre action "view" prend un paramètre : l'ID du post que nous aimerions voir. Ce paramètre est transmis à l'action grâce à l'URL demandée. Si un utilisateur demande /posts/view/3, alors la valeur '3' est transmise à la variable `$id`.

Nous faisons aussi une petite vérification d'erreurs pour nous assurer qu'un utilisateur accède bien à l'enregsitrement. Si un utilisateur requête `/posts/view`, nous lancerons un `NotFoundException` et laisserons le Gestionnaire d'Erreur de CakePHP ErrorHandler prendre le dessus. Nous executons aussi une vérification similaire pour nous assurer que l'utilisateur a accède à un enregistrement qui existe.

Maintenant, créons la vue pour notre nouvelle action "view" et plaçons-la dans `/app/View/Posts/view.ctp`.

```
<!-- Fichier : /app/View/Posts/view.ctp -->

<h1><?php echo h($post['Post']['title']); ?></h1>

<p><small>Créé le : <?php echo $post['Post']['created']; ?></small></p>

<p><?php echo h($post['Post']['body']); ?></p>
```

Vérifiez que cela fonctionne en testant les liens de la page /posts/index ou en affichant manuellement un post via `/posts/view/1`.

Ajouter des Posts

Lire depuis la base de données et nous afficher les posts est un bon début, mais lançons-nous dans l'ajout de nouveaux posts.

Premièrement, commençons par créer une action `add()` dans le PostsController :

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');
    public $components = array('Session');

    public function index() {
        $this->set('posts', $this->Post->find('all'));
    }

    public function view($id) {
        if (!$id) {
            throw new NotFoundException(__('Invalid post'));
        }

        $post = $this->Post->findById($id);
        if (!$post) {
```

```
            throw new NotFoundException(__('Invalid post'));
        }
        $this->set('post', $post);
    }

    public function add() {
        if ($this->request->is('post')) {
            $this->Post->create();
            if ($this->Post->save($this->request->data)) {
                $this->Session->setFlash(__('Votre post a été sauvegardé.'));
                $this->redirect(array('action' => 'index'));
            } else {
                $this->Session->setFlash(__('Impossible d\'ajouter votre post.'));
            }
        }
    }
}
```

Note : Vous avez besoin d'inclure le component Session (SessionComponent) et le helper Session (SessionHelper) dans chaque controller que vous utiliserez. Si nécessaire, incluez-les dans le controller principal (AppController) pour qu'ils soient accessibles à tous les controllers.

Voici ce que fait l'action `add()` : si la requête HTTP est de type POST, essayez de sauvegarder les données en utilisant le model "Post". Si pour une raison quelconque, la sauvegarde a échouée, affichez simplement la vue. Cela nous donne une chance de voir les erreurs de validation de l'utilisateur et d'autres avertissements.

Chaque requête de CakePHP contient un objet `CakeRequest` qui est accessible en utilisant `$this->request`. Cet objet contient des informations utiles sur la requête qui vient d'être reçue, et permet de contrôler les flux de votre application. Dans ce cas, nous utilisons la méthode `CakeRequest::is()` ` pour vérifier que la requête est de type POST.

Lorsqu'un utilisateur utilise un formulaire pour poster des données dans votre application, ces informations sont disponibles dans `$this->request->data`. Vous pouvez utiliser les fonctions `pr()` ou `debug()` pour les afficher si vous voulez voir à quoi cela ressemble.

Nous utilisons la méthode `SessionComponent::setFlash()` du component Session (SessionComponent) pour définir un message dans une variable session et qui sera affiché dans la page juste après la redirection. Dans le layout, nous trouvons la fonction `SessionHelper::flash` qui permet d'afficher et de nettoyer la variable correspondante. La méthode `Controller::redirect` ` du controller permet de rediriger vers une autre URL. Le paramètre `array('action' => 'index')` sera traduit vers l'URL /posts, c'est à dire l'action "index" du controller "Posts" (PostsController). Vous pouvez vous référer à l'API de la fonction `Router::url()` ` pour voir les différents formats d'URL acceptés dans les différentes fonctions de Cake.

L'appel de la méthode `save()` vérifiera les erreurs de validation et interrompra l'enregistrement s'il y en a une qui survient. Nous verrons la façon dont les erreurs sont traitées dans les sections suivantes.

Valider les données

Cake place la barre très haute pour briser la monotonie de la validation des champs de formulaires. Tout le monde déteste le dévelopement de formulaires interminables et leurs routines de validations. Cake rend tout cela plus facile et plus rapide.

Pour tirer profit des fonctionnalités de validation, vous devez utiliser le helper "Form" (FormHelper) dans vos vues. `FormHelper` est disponible par défaut dans toutes les vues avec la variables `$this->Form`.

Voici le code de notre vue "add" (ajout)

```
<!-- Fichier : /app/View/Posts/add.ctp -->

<h1>Ajouter un post</h1>
<?php
echo $this->Form->create('Post');
echo $this->Form->input('title');
echo $this->Form->input('body', array('rows' => '3'));
echo $this->Form->end('Sauvegarder le post');
?>
```

Nous utilisons ici le `FormHelper` pour générer la balise d'ouverture d'une formulaire HTML. Voici le code HTML généré par `$this->Form->create()` :

```
.. code-block:: html
```

<form id="PostAddForm" method="post" action="/posts/add">

Si `create()` est appelée sans aucun paramètre, Cake suppose que vous construisez un formulaire qui envoie les données en POST à l'action `add()` (ou `edit()` quand `id` est dans les données du formulaire) du controller actuel.

La méthode `$this->Form->input()` est utilisée pour créer des élements de formulaire du même nom. Le premier paramètre dit à CakePHP à quels champs ils correspondent et le second paramètre vous permet de spécifier un large éventail d'options - dans ce cas, le nombre de lignes du textarea. Il y a un peu d'introspection et "d'automagie" ici : `input()` affichera différents éléments de formulaire selon le champ spécifié du model.

L'appel de la méthode `$this->Form->end()` génère un bouton de soumission et ajoute la balise de fermeture du formulaire. Si une chaîne de caractères est passée comme premier paramètre de la méthode `end()`, le helper "Form" affichera un bouton de soumission dont le nom correspond à celle-ci. Encore une fois, référez-vous au chapitre *Helpers (Assistants)* pour en savoir plus sur les helpers.

A présent, revenons en arrière et modifions notre vue `/app/View/Posts/index.ctp` pour ajouter un lien "Ajouter un post". Ajoutez la ligne suivante avant `<table>`

```
<?php echo $this->Html->link(
    'Ajouter un Post',
    array('controller' => 'posts', 'action' => 'add')
); ?>
```

Vous vous demandez peut-être : comment je fais pour indiquer à CakePHP mes exigences de validation ? Les règles de validation sont définies dans le model. Retournons donc à notre model Post et précédons à quelques ajustements

```
class Post extends AppModel {
    public $validate = array(
        'title' => array(
            'rule' => 'notEmpty'
        ),
        'body' => array(
            'rule' => 'notEmpty'
        )
    );
}
```

Le tableau `$validate` indique à CakePHP comment valider vos données lorsque la méthode `save()` est appelée. Ici, j'ai spécifié que les deux champs "body" et "title" ne doivent pas être vides. Le moteur de validation de CakePHP est puissant, il dispose d'un certain nombre de règles pré-fabriquées (code de carte bancaire, adresse emails, etc.) et d'une souplesse pour ajouter vos propres règles de validation. Pour plus d'informations sur cette configuration, consultez le chapitre *Validation des Donnnées*.

Maintenant que vos règles de validation sont en place, utilisez l'application pour essayer d'ajouter un post avec un titre et un contenu vide afin de voir comment cela fonctionne. Puisque que nous avons utilisé la méthode `FormHelper::input()` ' du helper "Form" pour créer nos éléments de formulaire, nos messages d'erreurs de validation seront affichés automatiquement.

Editer des Posts

L'édition de posts : nous y voilà. Vous êtes un pro de CakePHP maintenant, vous devriez donc avoir adopté le principe. Créez d'abord l'action puis la vue. Voici à quoi l'action `edit()` du controller Posts (PostsController) devrait ressembler

```
public function edit($id = null) {
    if (!$id) {
        throw new NotFoundException(__('Invalid post'));
    }

    $post = $this->Post->findById($id);
    if (!$post) {
        throw new NotFoundException(__('Invalid post'));
    }

    if ($this->request->is('post') || $this->request->is('put')) {
        $this->Post->id = $id;
        if ($this->Post->save($this->request->data)) {
            $this->Session->setFlash(__('Your post has been updated.'));
            $this->redirect(array('action' => 'index'));
        } else {
            $this->Session->setFlash(__('Unable to update your post.'));
        }
    }
```

```
    if (!$this->request->data) {
        $this->request->data = $post;
    }
}
```

Cette action s'assure d'abord que l'utilisateur a essayé d'accéder à un enregistrement existant. Si il n'y a pas de paramètre `$id``passé, ou si le post n'existe pas, nous lançons une ``NotFoundException` pour que le gestionnaire d'Erreurs ErrorHandler de CakePHP s'en occupe.

Ensuite l'action vérifie que la requête est une requête POST. Si elle l'est, alors nous utilisons les données POST pour mettre à jour notre enregistrement Post, ou sortir et montrer les erreurs de validation à l'utilisateur.

Si il n'y a pas de données définies dans `$this->request->data`, nous le définissons simplement dans le post récupéré précédemment.

La vue d'édition devrait ressembler à quelque chose comme cela :

```
<!-- Fichier: /app/View/Posts/edit.ctp -->

<h1>Editer le post</h1>
<?php
    echo $this->Form->create('Post', array('action' => 'edit'));
    echo $this->Form->input('title');
    echo $this->Form->input('body', array('rows' => '3'));
    echo $this->Form->input('id', array('type' => 'hidden'));
    echo $this->Form->end('Sauvegarder le post');
```

Cette vue affiche le formulaire d'édition (avec les données pré-remplies) avec les messages d'erreur de validation nécessaires.

Une chose à noter ici : CakePHP supposera que vous éditez un model si le champ 'id' est présent dans le tableau de données. S'il n'est pas présent (ce qui revient à notre vue "add"), Cake supposera que vous insérez un nouveau model lorsque `save()` sera appelée.

Vous pouvez maintenant mettre à jour votre vue "index" avec des liens pour éditer des posts :

```
<!-- Fichier: /app/View/Posts/index.ctp  (lien d\'édition ajouté) -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link("Ajouter un Post", array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Titre</th>
        <th>Action</th>
        <th>Créé le</th>
    </tr>

<!-- Ici se trouve la boucle de notre tableau $posts, impression de l\'info du post -->

<?php foreach ($posts as $post): ?>
    <tr>
```

```
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'], array('action' => 'view',
        </td>
        <td>
            <?php echo $this->Html->link('Edit', array('action' => 'edit', $post['Post']['i
        </td>
        <td>
            <?php echo $post['Post']['created']; ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Supprimer des Posts

A présent, mettons en place un moyen de supprimer les posts pour les utilisateurs. Démarrons avec une action `delete()` dans le controller Posts (PostsController) :

```
public function delete($id) {
    if ($this->request->is('get')) {
        throw new MethodNotAllowedException();
    }
    if ($this->Post->delete($id)) {
        $this->Session->setFlash(__('Le Post avec l\'id ' . $id . ' a été supprimé.'));
        $this->redirect(array('action' => 'index'));
    }
}
```

Cette logique supprime le Post spécifié par $id, et utilise `$this->Session->setFlash()` pour afficher à l'utilisateur un message de confirmation après l'avoir redirigé sur `/posts`. Si l'utilisateur tente une suppression en utilisant une requête GET, une exeception est levée. Les exceptions manquées sont capturées par le gestionnaire d'exceptions de CakePHP et un joli message d'erreur est affiché. Il y a plusieurs *Exceptions* intégrées qui peuvent être utilisées pour indiquer les différentes erreurs HTTP que votre application pourrait rencontrer.

Etant donné que nous exécutons juste un peu de logique et de redirection, cette action n'a pas de vue. Vous voudrez peut-être mettre à jour votre vue "index" avec des liens pour permettre aux utilisateurs de supprimer des Posts, ainsi :

```
<!-- Fichier: /app/View/Posts/index.ctp -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link('Ajouter un Post', array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Titre</th>
        <th>Actions</th>
        <th>Créé le</th>
```

```
    </tr>

<!-- Ici, nous bouclons sur le tableau $post afin d'afficher les informations des posts -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'], array('action' => 'view',
        </td>
        <td>
            <?php echo $this->Form->postLink(
                'Delete',
                array('action' => 'delete', $post['Post']['id']),
                array('confirm' => 'Etes-vous sûr ?'));
            ?>
            <?php echo $this->Html->link('Editer', array('action' => 'edit', $post['Post']
        </td>
        <td>
            <?php echo $post['Post']['created']; ?>
        </td>
    </tr>
    <?php endforeach; ?>

</table>
```

Utiliser `postLink()` permet de créer un lien qui utilise du Javascript pour supprimer notre post en faisant une requête POST. Autoriser la suppression par une requête GET est dangereux à cause des robots d'indexation qui peuvent tous les supprimer.

Note : Ce code utilise aussi le helper "Form" pour demander à l'utilisateur une confirmation avant de supprimer le post.

Routes

Pour certains, le routage par défaut de CakePHP fonctionne suffisamment bien. Les développeurs qui sont sensibles à la facilité d'utilisation et à la compatibilité avec les moteurs de recherches apprécieront la manière dont CakePHP lie des URLs à des actions spécifiques. Nous allons donc faire une rapide modification des routes dans ce tutoriel.

Pour plus d'informations sur les techniques de routages, consultez le chapitre *Configuration des Routes*.

Par défaut, CakePHP effectue une redirection d'une personne visitant la racine de votre site (i.e. http ://www.exemple.com) vers le controller Pages (PagesController) et affiche le rendu de la vue appelée "home". Au lieu de cela, nous voudrions la remplacer avec notre controller Posts (PostsController).

Le routage de Cake se trouve dans `/app/Config/routes.php`. Vous devrez commenter ou supprimer la ligne qui définit la route par défaut. Elle ressemble à cela

```
Router::connect('/', array('controller' => 'pages', 'action' => 'display', 'home'));
```

Cette ligne connecte l'URL '/' à la page d'accueil par défaut de CakePHP. Nous voulons que cette URL soit connectée à notre propre controller, remplacez donc la ligne par celle-ci

```
Router::connect('/', array('controller' => 'posts', 'action' => 'index'));
```

Cela devrait connecter les utilisateurs demandant '/' à l'action `index()` de notre controller Posts (PostsController).

Note : CakePHP peut aussi faire du 'reverse routing' (ou routage inversé). Par exemple, pour la route définie plus haut, en ajoutant `array('controller' => 'posts', 'action' => 'index')` à la fonction retournant un tableau, l'URL '/' sera utilisée. Il est d'ailleurs bien avisé de toujours utiliser un tableau pour les URLs afin que vos routes définissent où vont les URLs, mais aussi pour s'assurer qu'elles aillent dans la même direction.

Conclusion

Créer des applications de cette manière vous apportera, paix, honneur, amour et argent au-delà même de vos fantasmes les plus fous. Simple n'est ce pas ? Gardez à l'esprit que ce tutoriel était très basique. CakePHP a *beaucoup* plus de fonctionnalités à offrir et il est aussi souple dans d'autres domaines que nous n'avons pas souhaité couvrir ici pour simplifier les choses. Utilisez le reste de ce manuel comme un guide pour développer des applications plus riches en fonctionnalités.

Maintenant que vous avez créé une application Cake basique, vous êtes prêt pour les choses sérieuses. Commencez votre propre projet et lisez le reste du Manuel et de l'API (http ://api20.cakephp.org).

Si vous avez besoin d'aide, venez nous voir sur le canal IRC #cakephp. Bienvenue sur CakePHP !

Prochaines lectures suggérées

Voici les différents chapitres que les gens veulent souvent lire après :

1. *Layouts* : Personnaliser les Layouts de votre application.
2. *Elements* : Inclure et ré-utiliser les portions de vues.
3. *Scaffolding* : Construire une ébauche d'application sans avoir à coder.
4. *Génération de code avec Bake* Générer un code CRUD basique.
5. *Authentification Simple et Autorisation de l'Application* : Tutoriel sur l'enregistrement et la connexion d'utilisateurs.

Lectures supplémentaires

Une requête CakePHP typique

Nous avons découvert les ingrédients de bases de CakePHP, regardons maintenant comment chaque objet travaille avec les autres pour répondre à une requête simple. Poursuivons sur notre exemple original

de requête, imaginons que notre ami Ricardo vient de cliquer sur le lien "Achetez un Cake personnalisé maintenant !" sur la page d'accueil d'une application CakePHP.

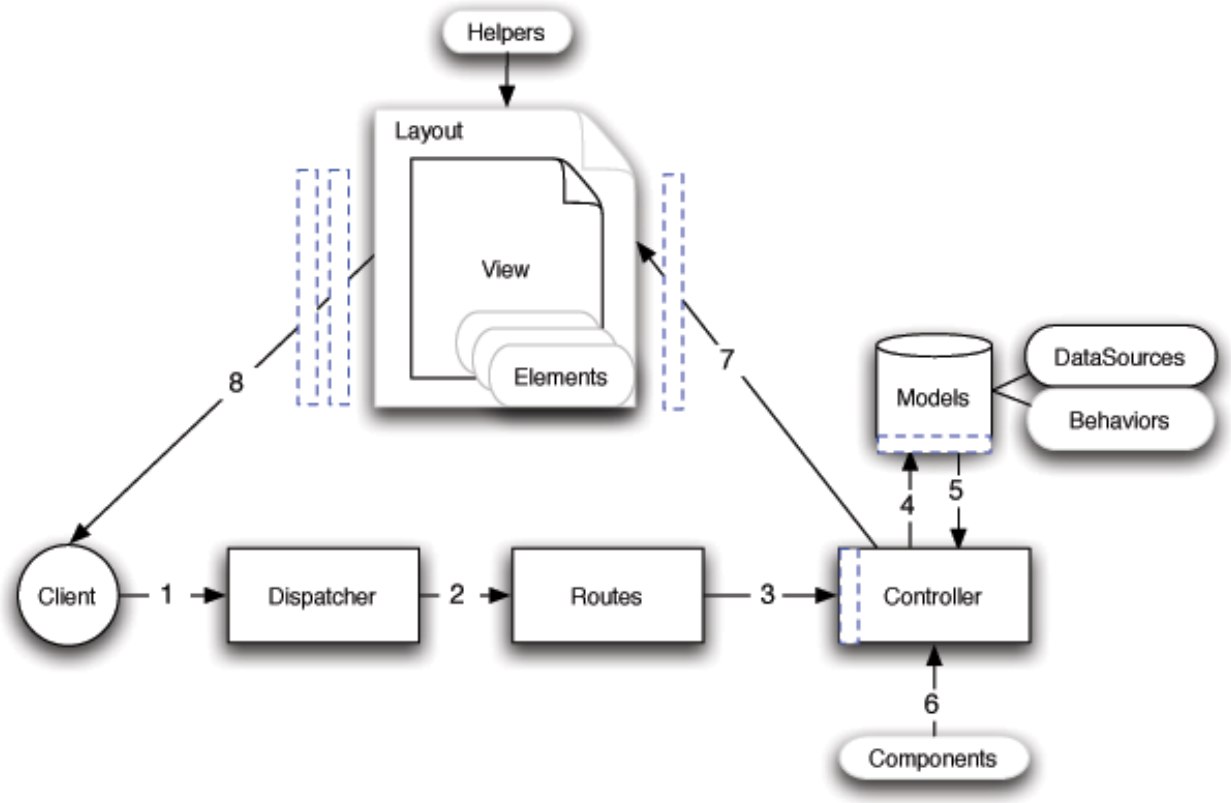

FIGURE 1.1 – Diagramme représentant une requête CakePHP typique.

Figure : 2. Typical Cake Request.

Noir = élément obligatoire, Gris = élément optionnel, Bleu = rappel (callback)

1. Ricardo clique sur le lien pointant vers http ://www.exemple.com/cakes/buy et son navigateur envoie une requête au serveur Web.
2. Le routeur analyse l'URL afin d'extraire les paramètres de cette requête : le controller, l'action et tout argument qui affectera la logique métier pendant cette requête.
3. En utilisant les routes, l'URL d'une requête est liée à une action d'un controller (une méthode d'une classe controller spécifique). Dans notre exemple, il s'agit de la méthode buy() du Controller Cakes. La fonction de rappel du controller, beforeFilter(), est appelée avant que toute logique de l'action du controller ne soit exécutée.
4. Le controller peut utiliser des models pour accéder aux données de l'application. Dans cet exemple, le controller utilise un model pour récupérer les derniers achats de Ricardo depuis la base de données. Toute méthode de rappel du model, tous behaviors ou sources de données peuvent s'appliquer pendant cette opération. Bien que l'utilisation du model ne soit pas obligatoire, tous les controllers CakePHP nécessitent au départ, au moins un model.
5. Une fois que le model a récupéré les données, elles sont retournées au controller. Des fonctions de rappel du model peuvent s'exécuter.

6. Le controller peut faire usage de components pour affiner les données ou pour effectuer d'autres opérations (manipulation de session, authentification ou envoi de mails par exemple).
7. Une fois que le controller a utilisé les models et components pour préparer suffisamment les données, ces données sont passées à la vue grâce à la méthode set(). Les méthodes de rappel (callbacks) du controller peuvent être appliquées avant l'envoi des données. La logique de la vue est exécutée, laquelle peut inclure l'utilisation d'éléments et/ou d'assistants. Par défaut, la vue est rendue à travers une mise en page (layout).
8. D'autres fonctions de rappel (callbacks) du controller (comme afterFilter) peuvent être exécutées. La vue complète et finale est envoyée au navigateur de Ricardo.

Conventions de CakePHP

Nous sommes de grands fans des conventions plutôt que de la configuration. Bien que cela réclame un peu de temps pour apprendre les conventions de CakePHP, à terme vous gagnerez du temps : en suivant les conventions, vous aurez des fonctionnalités automatiques et vous vous libérerez du cauchemar de la maintenance par l'analyse des fichiers de configuration. Les conventions sont aussi là pour créer un environnement de développement uniforme, permettant à d'autres développeurs de s'investir dans le code plus facilement.

Les conventions de CakePHP ont été créées à partir de nombreuses années d'expérience dans le développement Web et de bonnes pratiques. Alors que nous vous conseillons d'utiliser ces conventions lors de vos développements CakePHP, nous devons mentionner que la plupart de ces principes sont facilement contournables - ce qui est particulièrement utile lorsque vous travaillez avec d'anciennes applications.

Les conventions des Controllers

Les noms des classes de controller sont au pluriel, CamelCased et se terminent par `Controller`. `PeopleController` et `LatestArticlesController` sont des exemples respectant cette convention.

La première méthode que vous écrivez pour un controller devrait être `index()`. Lorsqu'une requête pointe vers un controller sans action, le comportement par défaut de CakePHP est d'exécuter la fonction `index()` de ce controller. Ainsi, la requête http ://www.exemple.com/apples/ renvoie à la fonction `index()` de `ApplesController`, alors que http ://www.exemple.com/apples/view renvoie vers la fonction `view()` de `ApplesController`.

Vous pouvez aussi changer la visibilité des méthodes des controllers dans CakePHP en préfixant les noms de méthode des controllers avec des underscores. Si une méthode du controller a été préfixée avec un underscore, la méthode ne sera pas accessible directement à partir du web mais est disponible pour une utilisation interne. Par exemple :

```
class NewsController extends AppController {

    public function latest() {
        $this->_findNewArticles();
    }

    protected function _findNewArticles() {
        // Logique pour trouver les derniers articles de nouvelles
```

```
    }
}
```

Alors que la page http ://www.exemple.com/news/latest/ est accessible pour l'utilisateur comme d'habitude, quelqu'un qui essaie d'aller sur la page http ://www.example.com/news/_findNewArticles/ aura une erreur, car la méthode est précédée d'un underscore. Vous pouvez aussi utiliser les mots-clés de visibilité de PHP pour indiquer si la méthode peut ou non être accessible à partir d'une url. Les méthodes non-publiques ne sont pas accessibles.

Considérations URL pour les noms de Controller Comme vous venez de voir, un controller à mot unique renvoie facilement vers un chemin URL en minuscules. Par exemple, `ApplesController` (qui serait défini dans le nom de fichier 'ApplesController.php') est accessible à l'adresse http ://exemple.com/apples.

Les controllers à multiples mots *peuvent* être de forme 'inflecté' qui correspondent au nom du controller :
- /redApples
- /RedApples
- /Red_apples
- /red_apples

iront tous vers l'index du controller RedApples. Cependant, la convention est que vos urls soient en minuscules et avec des underscores, c'est pourquoi /red_apples/go_pick est la forme correcte pour accéder à l'action `RedApplesController::go_pick`.

Pour plus d'informations sur les URLs de CakePHP et la gestion des paramètres, allez voir *Configuration des Routes*.

Conventions des Fichiers et des Noms de Classe

En général, les noms de fichiers correspondent aux noms des classes c'est-à-dire en CamelCase. Donc si vous avez une classe **MaChouetteClasse**, alors dans Cake, le fichier devra être nommé **MaChouetteClasse.php**. Voici des exemples de la manière dont on nomme les fichiers, pour chacun des différents types de classes que vous utiliseriez habituellement dans une application CakePHP :
- La classe controller **BisousEtCalinsController** devra se trouver dans un fichier nommé **BisousEtCalinsController.php**.
- La classe Component (Composant) **MonSuperComponent** devra se trouver dans un fichier nommé **MonSuperComponent.php**.
- La classe Model **ValeurOption** devra se trouver dans un fichier nommé **ValeurOption.php**.
- La classe Behavior (Comportement) **SpecialementFunkableBehavior** devra se trouver dans un fichier nommé **SpecialementFunkableBehavior.php**.
- La classe View (Vue) **SuperSimpleView** devra se trouver dans un fichier nommé **SuperSimpleView.ctp**.
- La classe Helper (Assistant) **LeMeilleurQuiSoitHelper** devra se trouver dans un fichier nommé **LeMeilleurQuiSoitHelper.php**.

Chaque fichier sera situé dans le répertoire approprié dans votre dossier app.

Conventions pour les Models et les Sources de données

Les noms de classe de model sont au singulier et en CamelCase. Person, BigPerson et ReallyBigPerson en sont des exemples.

Les noms de tables correspondant aux models CakePHP sont au pluriel et utilisent le caractère souligné (underscore). Les tables correspondantes aux models mentionnés ci-dessus seront donc respectivement : `people`, `big_people`, et `really_big_people`.

Note des traducteurs francophones : seul le dernier mot est au pluriel et tous les pluriels français ne seront pas compris par CakePHP sans lui indiquer précisément (par exemple cheval/chevaux). Voir pour cela le chapitre sur les inflexions.

Pour vous assurer de la syntaxe des mots pluriels et singuliers, vous pouvez utiliser la bibliothèque utilitaire `Inflector`. Voir la documentation sur *Inflector* pour plus d'informations.

Les noms des champs avec deux mots ou plus doivent être avec des underscores comme ici : first_name.

Les clés étrangères des relations hasMany, belongsTo ou hasOne sont reconnues par défaut grâce au nom (singulier) du model associé, suivi de _id. Donc, si un Cuisinier hasMany Cake, la table "cakes" se référera à un cuisinier de la table cuisiniers via une clé étrangère cuisinier_id. Pour une table avec un nom de plusieurs mots comme "type_categories", la clé étrangère sera type_categorie_id.

Les tables de jointure utilisées dans les relations hasAndBelongsToMany (HABTM) entre models doivent être nommées d'après le nom des tables des models qu'elles unissent, dans l'ordre alphabétique ("pommes_zebres" plutôt que "zebres_pommes").

Toutes les tables avec lesquelles les models de CakePHP interagissent (à l'exception des tables de jointure), nécessitent une clé primaire simple pour identifier chaque ligne de manière unique. Si vous souhaitez modéliser une table qui n'a pas de clé primaire sur un seul champ, la convention de CakePHP veut qu'une clé primaire sur un seul champ soit ajoutée à la table.

CakePHP n'accepte pas les clés primaires composées. Dans l'éventualité où vous voulez manipuler directement les données de votre table de jointure, cela veut dire que vous devez soit utiliser les appels directs à *query*, soit ajouter une clé primaire pour être en mesure d'agir sur elle comme un model normal. Exemple :

> CREATE TABLE posts_tags (id INT(10) NOT NULL AUTO_INCREMENT, post_id INT(10) NOT NULL, tag_id INT(10) NOT NULL, PRIMARY KEY(id)) ;

Plutôt que d'utiliser une clé auto-incrémentée comme clé primaire, vous pouvez aussi utiliser un champ char(36). Cake utilisera alors un uuid de 36 caractères (String : :uuid) lorsque vous sauvegardez un nouvel enregistrement en utilisant la méthode Model : :save.

Conventions des vues

Les fichiers de vue sont nommés d'après les fonctions du controller qu'elles affichent, sous une forme avec underscores. La fonction soyezPret() de la classe PersonnesController cherchera un gabarit de vue dans : /app/View/Personnes/soyez_pret.ctp.

Le schéma classique est /app/View/Controller/nom_de_fonction_avec_underscore.ctp.

En utilisant les conventions CakePHP dans le nommage des différentes parties de votre application, vous gagnerez des fonctionnalités sans les tracas et les affres de la configuration. Voici un exemple récapitulant les conventions abordées :

> Nom de la table dans la base de données : "personnes" Classe du Model : "Personne", trouvée dans /app/Model/Personne.php Classe du Controller : "PersonnesController", trouvée dans /app/Controller/PersonnesController.php Gabarit de la Vue : trouvé dans /app/View/Personnes/index.ctp

En utilisant ces conventions, CakePHP sait qu'une requête de type http ://exemple.com/personnes/ sera liée à un appel à la fonction index() du Controller PersonnesController, dans lequel le model Personne est automatiquement disponible (et automatiquement lié à la table 'personnes' dans la base) et rendue dans un fichier. Aucune de ces relations n'a été configurée par rien d'autre que la création des classes et des fichiers dont vous aviez besoin de toutes façons.

Maintenant que vous avez été initié aux fondamentaux de CakePHP, vous devriez essayer de dérouler le tutoriel du Blog CakePHP *Tutoriel d'un Blog* pour voir comment les choses s'articulent.

Structure du dossier de CakePHP

Après avoir téléchargé et extrait CakePHP, voici les fichiers et répertoires que vous devriez voir :
- app
- lib
- vendors
- plugins
- .htaccess
- index.php
- README

Vous remarquerez trois dossiers principaux :
- Le dossier *app* sera celui où vous exercerez votre magie : c'est là que vous placerez les fichiers de votre application.
- Le dossier *lib* est l'endroit où nous avons exercé notre propre magie. Engagez-vous personnellement à ne **pas** modifier les fichiers dans ce dossier. Nous ne pourrons pas vous aider si vous avez modifié le cœur du framework.
- Enfin, le dossier *vendors* est l'endroit où vous placerez vos librairies PHP tierces dont vous avez besoin pour vos applications CakePHP.

Le dossier App

Le répertoire app de CakePHP est l'endroit où vous réaliserez la majorité du développement de votre application. Regardons de plus près le contenu de ce répertoire.

Config Contient les (quelques) fichiers de configuration utilisés par CakePHP. Informations de connexion à la base de données, démarrage, fichiers de configuration de base et tous fichiers du même genre doivent être rangés ici.

Console Contient les commandes de la console et les Tasks de la console pour votre application. Ce répertoire peut aussi contenir un répertoire `Templates` pour personnaliser la sortie de bake. Pour plus d'informations, regardez `console-and-shells`.

Controller Contient vos controllers et leurs components.

Lib Contient les librairies internes qui ne proviennent pas de librairies externes. Cela vous permet de séparer les librairies internes de votre organisme des librairies externes.

Locale Stocke les fichiers pour l'internationalisation.

Model Pour les models, behaviors et sources de données de votre application.

Plugin Contient les packages des Plugins.

Test Ce répertoire contient tous les cas de test, et les fixtures de test pour votre application. Le répertoire `Test/Case` devra refléter votre application et contenir un ou plusieurs cas de test par classe dans votre application. Pour plus d'informations sur les cas de test et les fixtures de test, référez-vous à la documentation `development/testing`.

tmp C'est ici que CakePHP enregistre les données temporaires. La manière dont sont stockées les données actuelles dépend de la configuration que vous avez effectuée, mais ce répertoire est habituellement utilisé pour déposer les descriptions de models, les logs et parfois les informations de session.

Assurez vous que ce dossier existe et qu'il est en écriture, autrement la performance de votre application sera sévèrement impactée. En mode debug, CakePHP vous avertira si ce n'est pas le cas.

Vendor Toutes classes ou librairies tierces doivent être mises ici, de sorte qu'il sera facile d'y accéder par la fonction App : :import('vendor','name'). Les observateurs avisés noteront que cela semble redondant avec le répertoire "vendors" à la racine de l'arborescence. Nous aborderons les différences entre les deux lorsque nous discuterons de la gestion multi-applications et des configurations systèmes plus complexes.

View Les fichiers de présentation sont placés ici : éléments, pages d'erreur, helpers, layouts et vues.

webroot Dans un environnement de production, ce dossier doit être la racine de votre application. Les sous-répertoires sont utilisés pour les feuilles de style CSS, les images et les fichiers Javascript.

Structure de CakePHP

CakePHP dispose de classes de Controllers (Contrôleurs), de Models (Modèles), et de Views (Vues), mais il dispose de classes et objets supplémentaires qui rendent le développement en MVC plus rapide et amusant. Les Components (Composants), Behaviors (Comportements) et Helpers (Assistants) sont des classes qui offrent une extensibilité et une réutilisation, permettant d'ajouter rapidement des fonctionnalités aux classes MVC de base de vos applications. A ce stade de lecture, nous survolerons ces concepts, mais vous pourrez découvrir comment utiliser ces outils en détails plus tard.

Extensions de l'Application

Controllers, Helpers et Models ont chacun une classe parente, que vous pouvez utiliser pour définir des modifications impactant toute l'application. AppController (disponible dans `/app/Controller/AppController.php`), AppHelper (disponible dans `/app/View/Helper/AppHelper.php`) et AppModel (disponible dans `/app/Model/AppModel.php`) sont de bons choix pour écrire les méthodes que vous souhaitez partager entre tous vos controllers, helpers ou models.

Bien qu'elles ne soient pas une classe ou un fichier, les Routes jouent un rôle important dans les requêtes faites à CakePHP. La définition des routes indique à CakePHP comment lier les URLs aux actions des controllers. Le comportement par défaut suppose que l'URL `/controller/action/var1/var2` renvoie

vers `Controller::action($var1, $var2)` et à son action "action" qui prend deux paramètres ($var1, $var2). Mais vous pouvez utiliser les routes pour personnaliser les URLs et la manière dont elles sont interprétées par votre application.

Il peut être judicieux de regrouper certaines fonctionnalités. Un Greffon ou Plugin est un ensemble de models, de controllers et de vues qui accomplissent une tâche spécifique pouvant s'étendre à plusieurs applications. Un système de gestion des utilisateurs ou un blog simplifié pourraient être de bons exemples de plugins CakePHP.

Extensions du Controller ("Components")

Un Component (Composant) est une classe qui s'intègre dans la logique du controller. Si vos controllers ou vos applications doivent partager une logique, alors créer un Component est une bonne solution. A titre d'exemple, la classe intégrée EmailComponent rend triviale la création et l'envoi de courriels. Plutôt que d'écrire une méthode dans un seul controller qui effectue ce traitement, vous pouvez empaqueter ce code et ainsi le partager.

Les controllers sont également équipés de fonctions de rappel (callbacks). Ces fonctions sont à votre disposition au cas où vous avez besoin d'ajouter du code entre les différentes opérations internes de CakePHP. Les callbacks disponibles sont :
- `beforeFilter()`, exécutée avant toute action d'un controller.
- **`beforeRender()`, exécuté après le traitement du controller, mais avant** l'affichage de la vue.
- **`afterFilter()`, exécuté après la logique du controller, y compris** l'affichage de la vue. Il peut n'y avoir aucune différence entre `beforeRender()` et `afterFilter()`, à moins que vous n'ayez effectué un appel manuel à `render()` dans les actions de votre controller et que vous ayez inclus du code après cet appel.

Extensions du Model ("Behaviors")

De même, les Behaviors fonctionnent comme des passerelles pour ajouter une fonctionnalité commune aux models. Par exemple, si vous stockez les données d'un utilisateur dans une structure en arbre, vous pouvez spécifier que votre model Utilisateur se comporte comme un arbre, et il acquèrera automatiquement la capacité de suppression, d'ajout, et de déplacement des noeuds dans votre structure en arbre sous-jacente.

Les models sont aussi soutenus par une autre classe nommée une DataSource (source de données). Il s'agit d'une couche d'abstraction qui permet aux models de manipuler différents types de données de manière cohérente. La plupart du temps la source principale de données dans CakePHP est une base de données, vous pouvez cependant écrire des Sources de Données supplémentaires pour représenter des flux RSS, des fichiers CSV, des entrées LDAP ou des évènements iCal. Les Sources de Données vous permettent d'associer des enregistrements issus de sources différentes : plutôt que d'être limité à des jointures SQL, les Sources de Données vous permettent de dire à votre model LDAP qu'il est associé à plusieurs événements iCal.

Tout comme les controllers, les models sont également caractérisés par des fonctions de rappel (callbacks) :
- beforeFind()
- afterFind()
- beforeValidate()
- beforeSave()
- afterSave()

– beforeDelete()
– afterDelete()

Les noms de ces méthodes devraient être suffisamment explicites pour que vous compreniez leurs rôles. Vous obtiendrez plus de détails dans le chapître sur les models.

Extension de la Vue ("Helpers")

Un Helper (Assistant) est une classe d'assistance pour les vues. De même que les components sont utilisés par plusieurs controllers, les helpers permettent à différentes vues d'accéder et de partager une même logique de présentation. L'un des helpers intégrés à Cake, AjaxHelper, facilite les requêtes Ajax dans les vues.

La plupart des applications ont des portions de code pour les vues qui sont répétitives. CakePHP facilite la réutilisabilité de ce code grâce aux Layouts (mises en pages) et aux Elements. Par défaut, toutes les vues affichées par un controller ont le même layout. Les elements sont utilisés lorsque de petites portions de contenu doivent apparaître dans plusieurs vues.

CHAPITRE 2

Installation

CakePHP est rapide et facile à installer. Les conditions minimum requises sont un serveur web et une copie de Cake, c'est tout ! Bien que ce manuel se focalise principalement sur la configuration avec Apache (parce que c'est le cas le plus commun), vous pouvez configurer Cake pour lancer une diversité de serveurs web tels que LightHTTPD ou Microsoft IIS.

Conditions requises

– HTTP Server. Par exemple : Apache. mod_rewrite est préférable, mais en aucun cas nécessaire.
– PHP 5.2.8 ou plus.

Techniquement, un moteur de base de données n'est pas nécessaire, mais nous imaginons que la plupart des applications vont en utiliser un. CakePHP supporte une diversité de moteurs de stockage de données :
– MySQL (4 ou plus)
– PostgreSQL
– Microsoft SQL Server
– SQLite

Note : Les drivers intégrés requièrent tous PDO. Vous devez vous assurer que vous avez les bonnes extensions PDO installées.

Licence

CakePHP est licencié sous la licence MIT. Cela signifie que vous êtes libre de modifier, distribuer et reproduire le code source sous la condition que les informations de copyright restent intactes. Vous êtes aussi libres d'incorporer CakePHP dans toute code source d'application commerciale ou fermée.

Télécharger CakePHP

Il y a deux façons d'obtenir une copie récente de CakePHP. Vous pouvez soit télécharger une copie archivée de (zip/tar.gz/tar.bz2) à partir du site web principal, soit faire un check out du code sur dépôt de git.

Pour télécharger la dernière version majeure de CakePHP. Visitez le site web principal http ://cakephp.org et suivez le lien "Télécharger maintenant".

Toutes les versions actuelles de CakePHP sont hébergées sur Github (http ://github.com/cakephp/cakephp). Github héberge CakePHP lui-même ainsi que plusieurs autres plugins pour CakePHP. Les versions de CakePHP sont disponibles à Téléchargements Github (https ://github.com/cakephp/cakephp/tags).

Sinon, vous pouvez obtenir du code frais avec tous les correctifs de bug et à jour des améliorations de dernière minute. Ceux-ci peuvent être accessibles à partir de github en clonant le répertoire de Github (http ://github.com/cakephp/cakephp)

```
git clone git://github.com/cakephp/cakephp.git
```

Permissions

CakePHP utilise le répertoire `app/tmp` pour un certain nombre d'opérations. Les descriptions de Modèle, les vues mises en cache, et les informations de session sont juste quelques exemples.

De même, assurez-vous que le répertoire `app/tmp` et tous ses sous-répertoires dans votre installation cake sont en écriture pour l'utilisateur du serveur web.

Configuration

Configurer CakePHP est aussi simple que de le flanquer dans le document root de votre serveur web, ou aussi complexe et flexible que vous le souhaitez. Cette section couvrira les trois types principaux d'installation pour CakePHP : développement, production, et avancé.

- Développement : Facile à mettre en oeuvre, mais les URLs de l'application contiennent le nom du répertoire d'installation de CakePHP et c'est moins sécurisé.
- Production : Nécessite d'être habilité à configurer le Document Root du serveur, URLs propres, très sécurisé.
- Avancé : Avec un peu de configuration, vous permet de placer les répertoires clés de CakePHP à différents endroits du système de fichiers, avec la possibilité de partager un seul répertoire de la librairie centrale CakePHP entre plusieurs applications.

Développement

Une installation "développement" est la méthode la plus rapide pour lancer Cake. Cet exemple vous aidera à installer une application CakePHP et à la rendre disponible à http ://www.example.com/cake_2_0/. Nous considérons pour les besoins de cet exemple que votre document root pointe sur `/var/www/html`.

Décompressez le contenu de l'archive Cake dans `/var/www/html`. Vous avez maintenant un dossier dans votre document root, nommé d'après la version que vous avez téléchargée (par exemple : cake_2.0.0). Renommez ce dossier en "cake_2_0". Votre installation "développement" devrait ressembler à quelque chose comme cela dans votre système de fichiers :

```
/var/www/html/
    cake_2_0/
        app/
        lib/
        plugins/
        vendors/
        .htaccess
        index.php
        README
```

Si votre serveur web est configuré correctement, vous devriez trouver maintenant votre application Cake accessible à http ://www.exemple.com/cake_2_0/.

Utiliser un CakePHP pour de multiples applications

Si vous développez un certain nombre d'applications il peut sembler être sensé de partager le même coeur de CakePHP. Il y a peu de façon d'accomplir cela. Souvent, le plus facile est d'utiliser le `include_path` de PHP. Pour commencer, copiez CakePHP dans un répertoire. Pour cet exemple, nous utiliserons `~/projects` :

```
git clone git://github.com/cakephp/cakephp.git ~/projects/cakephp
```

Cela copiera CakePHP dans votre répertoire de `~/projects`. Si vous ne voulez pas utiliser git, vous pouvez télécharger un zipball et les étapes restantes seront les mêmes. Ensuite, vous devrez localiser et modifier votre `php.ini`. Sur les systèmes *nix, il se trouve souvent dans `/etc/php.ini`, mais en utilisant `php -i` et en regardant 'Loaded Configuration File' (Fichier de Configuration Chargé). Une fois que vous avez trouvé le bon fichier ini, modifier la configuration de `include_path` pour inclure `~/projects/cakephp/lib`. Un exemple ressemblerait à cela :

```
include_path = .:/home/mark/projects/cakephp/lib:/usr/local/php/lib/php
```

Après avoir redémarré votre serveur web, vous devriez voir les changements dans `phpinfo()`.

Note : Si vous êtes sur windows, les chemins d'inclusion sont séparés par des ; au lieu de :

Une fois que vous avez configuré votre `include_path`, vos applications devraient être capable de trouver automatiquement CakePHP.

Production

Une installation "production" est une façon plus flexible de lancer Cake. Utiliser cette méthode permet à tout un domaine d'agir comme une seule application CakePHP. Cet exemple vous aidera à installer Cake

n'importe où dans votre système de fichiers et à le rendre disponible à l'adresse : http ://www.exemple.com. Notez que cette installation demande d'avoir les droits pour modifier le `DocumentRoot` sur le serveur web Apache.

Décompressez les contenus de l'archive Cake dans un répertoire de votre choix. Pour les besoins de cet exemple, nous considérons que vous avez choisi d'installer Cake dans /cake_install. Votre installation de production devrait ressembler à quelque chose comme çà dans votre système de fichiers :

```
/cake_install/
    app/
        webroot/ (ce répertoire est défini comme répertoire
        ``DocumentRoot``)
    lib/
    plugins/
    vendors/
    .htaccess
    index.php
    README
```

Les développeurs utilisant Apache devraient régler la directive `DocumentRoot` pour le domaine à :

```
DocumentRoot /cake_install/app/webroot
```

Si votre serveur web est configuré correctement, vous devriez maintenant accéder à votre application Cake accessible à l'adresse : http ://www.exemple.com.

Installation avancée et URL Rewriting

Installation avancée

Installer CakePHP avec l'installeur PEAR

CakePHP publie un package PEAR que vous pouvez installer en utilisant l'installeur pear. L'installation avec l'installeur PEAR peut simplifier le partage des librairies de CakePHP sur plusieurs applications. Pour installer CakePHP avec pear, vous devrez faire comme suit :

```
pear channel-discover pear.cakephp.org
pear install cakephp/CakePHP
```

Note : Sur certains systèmes, l'installation de librairies avec pear nécessitera la commande `sudo`.

Après avoir installé CakePHP avec pear, si pear est configuré correctement, vous devriez pouvoir utiliser la commande `cake` pour créer une nouvelle application. Puisque CakePHP sera localisé dans l'`include_path` de PHP, vous n'aurez pas besoin de faire d'autres changements.

Installer CakePHP avec composer

Composer est un outil de gestion de dépendance pour PHP 5.3+. Il règle plusieurs problèmes que l'installeur PEAR a, et simplifie la gestion de plusieurs versions de librairies. Puisque CakePHP publie un package PEAR, vous pouvez installer CakePHP en utilisant composer (http ://getcomposer.org). Avant d'installer CakePHP, vous devrez configurer un fichier `composer.json`. Un fichier composer.json pour une application CakePHP ressemblerait à ce qui suit :

```
{
    "name": "example-app",
    "repositories": [
        {
            "type": "pear",
            "url": "http://pear.cakephp.org"
        }
    ],
    "require": {
        "pear-cakephp/cakephp": ">=2.3.4"
    },
    "config": {
        "vendor-dir": "Vendor/"
    }
}
```

Sauvegarder ce JSON dans `composer.json` dans le répetoire racine de votre projet. Ensuite, télécharger le fichier composer.phar dans votre projet. Après avoir téléchargé composer, installez CakePHP. Dans le même répertoire que votre fichier `composer.json`, lancez ce qui suit :

```
$ php composer.phar install
```

Une fois que composer a terminé son exécution, vous devriez avoir une structure de répertoire qui ressemble à :

```
example-app/
    composer.phar
    composer.json
    Vendor/
        bin/
        autoload.php
        composer/
        pear-pear.cakephp.org/
```

Vous êtes maintenant prêt à générer le reste du squelette de votre application :

```
$ Vendor/bin/cake bake project <path to project>
```

Par défaut `bake` va hard-code `CAKE_CORE_INCLUDE_PATH`. Pour rendre votre application plus portable, vous devrez modifier `webroot/index.php`, en changeant `CAKE_CORE_INCLUDE_PATH` en un chemin relatif :

```
define(
    'CAKE_CORE_INCLUDE_PATH',
    ROOT . DS . APP_DIR . '/Vendor/pear-pear.cakephp.org/CakePHP'
);
```

Si vous installez d'autres librairies avec composer, vous devrez configurer l'autoloader, et régler un problème dans l'autoloader de composer. Dans votre fichier `Config/bootstrap.php`, ajoutez ce qui suit :

```
// Charger l'autoload de composer.
require APP . '/Vendor/autoload.php';

// Retire et re-prepend l'autoloader de CakePHP puisque composer pense que c'est le plus i
// See https://github.com/composer/composer/commit/c80cb76b9b5082ecc3e5b53b1050f76bb27b127
spl_autoload_unregister(array('App', 'load'));
spl_autoload_register(array('App', 'load'), true, true);
```

Vous devriez maintenant avoir une application CakePHP fonctionnelle avec CakePHP installé via composer. Assurez-vous de garder les fichiers composer.json et composer.lock.json avec le reste de votre code source.

Partager les librairies de CakePHP pour plusieurs applications

Il peut y avoir des situations où vous voulez placer les répertoires de CakePHP à différents endroits du système de fichiers. Cela est peut être dû à des restrictions de l'hôte partagé, ou peut-être souhaitez-vous juste que quelques-unes de vos apps puissent partager les mêmes librairies de CakePHP. Cette section décrit comment déployer vos répertoires de CakePHP à travers le système de fichiers.

Premièrement, réalisez qu'il y a trois parties principales d'une application Cake :

1. Les librairies du coeur de CakePHP, dans /lib/Cake.
2. Le code de votre application, dans /app.
3. Le webroot de l'application, habituellement dans /app/webroot.

Chacun de ces répertoires peut être situé n'importe où dans votre système de fichier, avec l'exception de webroot, qui a besoin d'être acessible pour votre serveur web. Vous pouvez même déplacer le dossier webroot en-dehors du dossier app tant que vous dîtes à Cake où vous le mettez.

Pour configurer votre installation de Cake, vous aurez besoin de faire quelques changements aux fichiers suivants.
– /app/webroot/index.php
– /app/webroot/test.php (si vous utilisez la fonctionnalité de *Testing*.)
Il y a trois constantes que vous devrez modifier : `ROOT`, `APP_DIR`, et `CAKE_CORE_INCLUDE_PATH`.
– `ROOT` doit être définie vers le chemin du répertoire qui contient le dossier app.
– `APP_DIR` doit être définie vers le nom (de base) de votre dossier app.
– `CAKE_CORE_INCLUDE_PATH` doit être définie vers le chemin du dossier des librairies de CakePHP.
Testons cela avec un exemple pour que vous puissiez voir ce à quoi peut ressembler une installation avancée en pratique. Imaginez que je souhaite configurer CakePHP pour travailler comme ce qui suit :
– Les librairies du coeur de CakePHP seront placées dans /usr/lib/cake.
– Le répertoire webroot de l'application sera /var/www/monsite/.
– Le répertoire app de mon application sera /home/moi/monapp.

Etant donné ce type de configuration, j'aurais besoin de modifier mon fichier webroot/index.php (ce qui finira dans /var/www/mysite/index.php, dans cet exemple) pour ressembler à ce qui suit :

```
// /app/webroot/index.php (partiel, commentaires retirés)

if (!defined('ROOT')) {
    define('ROOT', DS.'home'.DS.'moi');
}

if (!defined('APP_DIR')) {
    define ('APP_DIR', 'monapp');
}

if (!defined('CAKE_CORE_INCLUDE_PATH')) {
    define('CAKE_CORE_INCLUDE_PATH', DS.'usr'.DS.'lib');
}
```

Il est recommandé d'utiliser la constante `DS` plutôt que des slashes pour délimiter des chemins de fichier. Cela empêche les erreurs de fichiers manquants que vous pourriez obtenir en résultats en utilisant le mauvais délimiteur, et cela rend votre code plus portable.

Apache et mod_rewrite (et .htaccess)

Cette section a été déplacée vers *URL rewriting*.

URL Rewriting

Apache et mod_rewrite (et .htaccess)

Alors que CakePHP est construit pour travailler avec mod_rewrite –et habituellement il l'est– nous avons remarqué que certains utilisateurs se battent pour obtenir un bon fonctionnement sur leurs systèmes.

Ici il y a quelques trucs que vous pourriez essayer pour que cela fonctionne correctement. Premièrement, regardez votre fichier httpd.conf (Assurez vous que vous avez éditer le httpd.conf du système plutôt que celui d'un utilisateur- ou le httpd.conf d'un site spécifique).

1. Assurez-vous qu'un .htaccess est permis et que AllowOverride est défini à All pour le DocumentRoot correct. Vous devriez voir quelque chose comme :

```
    # Chaque répertoire auquel Apache a accès peut être configuré avec
    # respect pour lesquels les services et les fonctionnalités sont
    # autorisés et/ou désactivés dans ce répertoire (et ses sous-répertoires).
    #
    # Premièrement, nous configurons "par défault" pour être un ensemble
    # très restrictif de fonctionnalités.
    #
    <Directory />
        Options FollowSymLinks
        AllowOverride All
    #    Order deny,allow
```

```
#    Deny from all
</Directory>
```

2. Assurez-vous que vous avez chargé correctement mod_rewrite. Vous devriez voir quelque chose comme :

```
LoadModule rewrite_module libexec/apache2/mod_rewrite.so
```

   Dans la plupart des systèmes, ceux-ci vont être commentés (en étant précédé par un #) par défaut, donc vous aurez juste besoin de retirer les symboles # du début.

   Après que vous avez fait des changements, re-démarrez Apache pour être sûr que les paramètres soient actifs.

   Vérifiez que vos fichiers .htaccess sont effectivement dans le bon répertoire.

   Cela peut arriver pendant la copie parce que certains systèmes d'exploitation traitent les fichiers qui commencent par '.' en caché et du coup ne les voient pas pour les copier.

3. Assurez-vous que votre copie de CakePHP vient de la section des téléchargements du site de notre dépôt GIT, et a été dézippé correctement en vérifiant les fichiers .htaccess.

   Le répertoire root de Cake (a besoin d'être copié dans votre document, cela redirige tout vers votre app Cake) :

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$ app/webroot/    [L]
   RewriteRule    (.*) app/webroot/$1 [L]
</IfModule>
```

   Le répertoire app de Cake (sera copié dans le répertoire supérieur de votre application avec Bake) :

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$    webroot/    [L]
   RewriteRule    (.*) webroot/$1    [L]
</IfModule>
```

   Le répertoire webroot de Cake (sera copié dans le webroot de votre application avec Bake) :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php/$1 [QSA,L]
</IfModule>
```

   Si votre site Cakephp a toujours des problèmes avec mod_rewrite, essayez de modifier les paramètres pour les virtualhosts. Si vous êtes sur ubuntu, modifiez le fichier /etc/apache2/sites-available/default (l'endroit dépend de la distribution). Dans ce fichier, assurez-vous que `AllowOverride None` a changé en `AllowOverride All`, donc vous avez :

```
<Directory />
    Options FollowSymLinks
    AllowOverride All
</Directory>
```

```
<Directory /var/www>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order Allow,Deny
    Allow from all
</Directory>
```

Si vous êtes sur Mac OSX, une autre solution est d'utiliser l'outil virtualhostx pour faire un hôte virtuel pour pointer vers votre dossier.

Pour beaucoup de services d'hébergement (GoDaddy, 1and1), votre serveur web est en fait déjà distribué à partir d'un répertoire utilisateur qui utilise déjà mod_rewrite. Si vous installez CakePHP dans un répertoire utilisateur (http ://exemple.com/~username/cakephp/), ou toute autre structure d'URL qui utilise déjà mod_rewrite, vous aurez besoin d'ajouter les requêtes (statements) RewriteBase aux fichiers .htaccess que CakePHP utilise (/.htaccess, /app/.htaccess, /app/webroot/.htaccess).

Ceci peut être ajouté à la même section avec la directive RewriteEngine, donc par exemple, votre fichier .htaccess dans webroot ressemblerait à ceci :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/cake/app
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php/$1 [QSA,L]
</IfModule>
```

Les détails de ces changements dépendront de votre configuration, et pourront inclure des choses supplémentaires qui ne sont pas liées à CakePHP. Merci de vous renseigner sur la documentation en ligne d'Apache pour plus d'informations.

4. (Optionel) Pour améliorer la configuration de production, vous devriez empêcher les assets invalides d'être parsés par CakePHP. Modifier votre webroot .htaccess pour quelque chose comme :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/cake/app
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_URI} !^/(app/webroot/)?(img|css|js)/(.*)$
    RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

Ce qui est au-dessus va simplement empêcher les assets incorrects d'être envoyés à index.php et à la place d'afficher la page 404 de votre serveur web.

De plus, vous pouvez créer une page HTML 404 correspondante, ou utiliser la page 404 de CakePHP intégrée en ajoutant une directive `ErrorDocument` :

```
ErrorDocument 404 /404-not-found
```

De belles URLs sur nginx

nginx est un serveur populaire qui, comme Lighttpd, utilise moins de ressources système. Son inconvénient est qu'il ne fait pas usage de fichiers .htaccess comme Apache et Lighttpd, il est donc nécessaire de créer

les URLs réécrites dans la configuration du site disponibles. selon votre configuration, vous devrez modifier cela, mais à tout le moins, vous aurez besoin de PHP fonctionnant comme une instance FastCGI.

```
server {
    listen   80;
    server_name www.example.com;
    rewrite ^(.*) http://example.com $1 permanent;
}

server {
    listen   80;
    server_name example.com;

    # root directive should be global
    root   /var/www/example.com/public/app/webroot/;
    index  index.php;

    access_log /var/www/example.com/log/access.log;
    error_log /var/www/example.com/log/error.log;

    location / {
        try_files $uri $uri/ /index.php?$uri&$args;
    }

    location ~ \.php $ {
        include /etc/nginx/fastcgi_params;
        try_files $uri =404;
        fastcgi_pass    127.0.0.1:9000;
        fastcgi_index   index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

Rewrites d'URL sur IIS7 (serveurs Windows)

IIS7 ne supporte pas nativement les fichiers .htaccess. Bien qu'il existe des add-ons qui peuvent ajouter ce support, vous pouvez aussi importer les règles des .htaccess dans IIS pour utiliser les rewrites natifs de CakePHP. Pour ce faire, suivez ces étapes :

1. Utilisez l'installeur de la plateforme Web de Microsoft (http ://www.microsoft.com/web/downloads/platform.aspx) pour installer l'URL Rewrite Module 2.0 (http ://www.iis.net/downloads/microsoft/url-rewrite) ou télécharger le directement (32-bit (http ://www.microsoft.com/en-us/download/details.aspx ?id=5747) / 64-bit (http ://www.microsoft.com/en-us/download/details.aspx ?id=7435)).
2. Créez un nouveau fichier dans votre dossier CakePHP, appelé web.config.
3. Utilisez Notepad ou tout autre éditeur XML-safe, copiez le code suivant dans votre nouveau fichier web.config...

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
```

```
            <rules>
                <rule name="Rewrite requests to test.php" stopProcessing="true">
                    <match url="^test.php(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="app/webroot/test.php{R:1}" />
                </rule>
                <rule name="Exclude direct access to app/webroot/*" stopProcessing="true">
                    <match url="^app/webroot/(.*)$" ignoreCase="false" />
                    <action type="None" />
                </rule>
                <rule name="Rewrite routed access to assets (img, css, files, js, favicon)"
                    <match url="^(img|css|files|js|favicon.ico)(.*)$" />
                    <action type="Rewrite" url="app/webroot/{R:1}{R:2}" appendQueryString="
                </rule>
                <rule name="Rewrite requested file/folder to index.php" stopProcessing="tru
                    <match url="^(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="index.php" appendQueryString="true" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

Une fois que le fichier web.config est créé avec les bonnes règles de réécriture des liens de IIS, les liens CakePHP, les CSS, les JS, et le reroutage devraient fonctionner correctement.

Je ne veux / ne peux utiliser l'URL rewriting

Si vous ne voulez ou ne pouvez pas utiliser l'URL rewriting sur votre serveur web, référez-vous à la section *core configuration*.

A vous de jouer !

Ok, voyons voir CakePHP en action. Selon la configuration que vous utilisez, vous pouvez pointer votre navigateur vers http ://exemple.com/ ou http ://exemple.com/cake_install/. A ce niveau, vous serez sur le page home par défaut de CakePHP, et un message qui vous donnera le statut de la connection de votre base de données courante.

Félicitations ! Vous êtes prêt à *créer votre première application CakePHP*.

Cela ne fonctionne pas ? Si vous avez une erreur liée au timezone de PHP, décommentez une ligne dans `app/Config/core.php` :

```
/**
 * Décommentez cette ligne et corrigez votre serveur de timezone pour régler
 * toute erreur liée à la date & au temps.
 */
    date_default_timezone_set('UTC');
```

CHAPITRE 3

Débuter avec CakePHP

Bienvenue dans le CookBook, le manuel du framework d'applications web, CakePHP. Avec CakePHP, développer c'est du gâteau !

Lire ce manuel suppose que vous ayez une connaissance générale de PHP et une connaissance de base de la programmation orientée-objet (POO). Certaines fonctionnalités livrées avec le framework entraînent l'utilisation de technologies différentes - comme SQL, JavaScript et XML - que ce manuel ne tente pas d'expliquer, il indique seulement de quelle manière elles sont utilisées dans ce contexte.

Qu'est ce que CakePHP ? Pourquoi l'utiliser ?

CakePHP (http ://www.cakephp.org/) est un framework (http ://en.wikipedia.org/wiki/Application_framework) pour PHP (http ://www.php.net/) gratuit (http ://en.wikipedia.org/wiki/MIT_License), open-source (http ://en.wikipedia.org/wiki/Open_source), de développement rapide (http ://en.wikipedia.org/wiki/Rapid_application_development). C'est une structure fondamentale pour les programmeurs pour créer des applications web. Notre principal objectif est de vous permettre de travailler d'une manière structurée et rapide -sans perte de flexibilité.

CakePHP rompt la monotonie du développement web. Nous vous offrons tous les outils nécessaires pour ne coder que ce dont vous avez réellement besoin : la logique spécifique de votre application.

Au lieu de réinventer la roue à chaque fois que vous démarrez un nouveau projet, récupérez une copie de CakePHP et concentrez-vous sur les « entrailles » de votre application.

CakePHP dispose d'une équipe de développement (http ://cakephp.lighthouseapp.com/contributors) et d'une communauté actives, qui donnent au projet une forte valeur ajoutée. En plus de vous éviter la ré-invention de la roue, l'utilisation de CakePHP implique que le coeur de votre application est bien testé et qu'il peut être constamment amélioré.

Voici un aperçu rapide des caractéristiques que vous apprécierez en utilisant CakePHP :
- Communauté active et sympathique (http ://cakephp.org/feeds).
- Système de licence souple (http ://en.wikipedia.org/wiki/MIT_License).
- Compatible avec les versions PHP 5.2.8 et supérieures.

- Fonctions CRUD (http ://en.wikipedia.org/wiki/Create,_read,_update_and_delete). (create, read, update, delete) intégrées pour les interactions avec la base de données.
- Scaffolding (http ://en.wikipedia.org/wiki/Scaffold_(programming)) (maquettage rapide) d'application.
- Génération de code.
- Architecture MVC (http ://en.wikipedia.org/wiki/Model-view-controller).
- Dispatcheur de requêtes avec des URLs propres et personnalisables grâce un système de routes.
- Validation intégrée des données (http ://en.wikipedia.org/wiki/Data_validation).
- Système de template (http ://en.wikipedia.org/wiki/Web_template_system) rapide et souple (syntaxe PHP avec des Helpers).
- Helpers (assistants) de vue pour AJAX, JavaScript, formulaires HTML...
- Components (composants) intégrés : Email, Cookie, Security, Session et Request Handling.
- Système de contrôle d'accès ACL (http ://en.wikipedia.org/wiki/Access_control_list) flexible.
- Nettoyage des données.
- Système de cache (http ://en.wikipedia.org/wiki/Web_cache) souple.
- Localisation et internationalisation.
- Fonctionne sur n'importe quelle arborescence de site web, avec un zest de configuration Apache (http ://httpd.apache.org/) pas très compliquée.

Comprendre le système M-V-C (Model-View-Controller)

CakePHP suit le motif de conception logicielle MVC (http ://en.wikipedia.org/wiki/Model-view-controller). Programmer en utilisant MVC sépare votre application en 3 couches principales :

La couche Model

La couche Model représente la partie de l'application qui exécute la logique métier. Cela signifie qu'elle est responsable de récuperer les données, de les convertir selon des concepts chargés de sens pour votre application, tels que le traitement, la validation, l'association et beaucoup d'autres tâches concernant la manipulation des données.

A première vue, l'objet Model peut être vu comme la première couche d'intéraction avec n'importe quelle base de données que vous pourriez utiliser pour votre application. Mais plus globalement, ils font partie des concepts majeurs autour desquels vous allez exécuter votre application.

Dans le cas d'un réseau social, la couche Model s'occupe des tâches comme de sauvegarder des données, de sauvegarder des associations d'amis, d'enregistrer et de récupérer les photos des utilisateurs, de trouver des suggestions de nouveaux amis, etc ... Tandis que les objets Models seront "Ami", "User", "Commentaire", "Photo".

Le couche Vue

La Vue retourne une présentation des données venant du model. Etant séparée par les Objets Model, elle est responsable de l'utilisation des informations dont elle dispose pour produire une interface de présentation de votre application.

Par exemple, de la même manière que la couche Model retourne un ensemble de données, la Vue utilise ces données pour fournir une page HTML les contenant. Ou un résultat XML formaté pour que d'autres l'utilisent.

La couche Vue n'est pas seulement limitée au HTML ou à la répresentation en texte de données. Elle peut aussi être utilisée pour offrir une grande variété de formats en fonction de vos besoins, comme les vidéos, la musique, les documents et tout autre format auquel vous pouvez penser.

La couche Controller

La couche Controller gère les requêtes des utilisateurs. Elle est responsable de retourner une réponse avec l'aide mutuelle des couches Model et Vue.

Les Controllers peuvent être imaginés comme des managers qui ont pour mission que toutes les ressources souhaitées pour accomplir une tâche soient déléguées aux travailleurs corrects. Il attend des pétitions des clients, vérifie leur validité selon l'authentification et les règles d'autorisation, délèguent les données récupérées et traitées par le Model, et sélectionne les type de présentation correctes que le client accepte, pour finalement déléguer le processus d'affichage à la couche Vue.

Cycle de la requête CakePHP

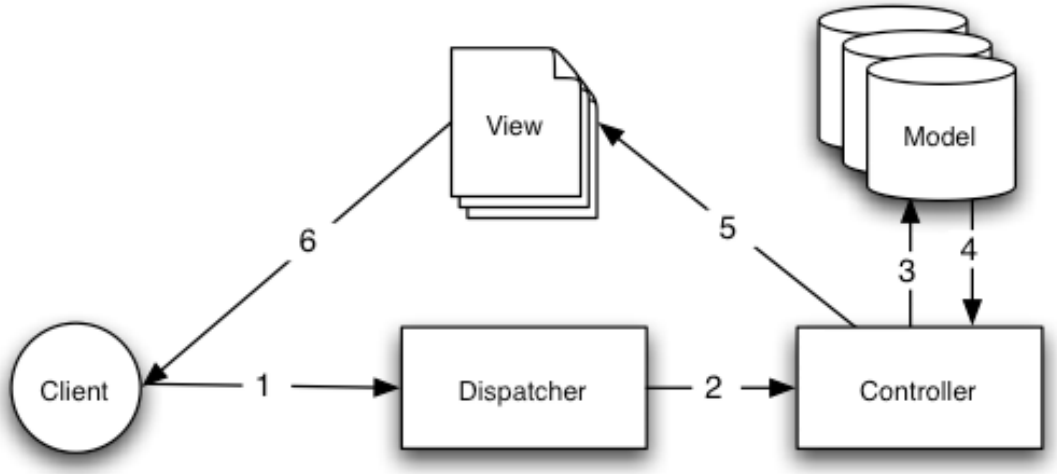

Figure : 1 : Une requête MVC basique

Figure : 1 Montre la gestion typique d'une requête client dans CakePHP

Le cycle d'une requête CakePHP typique débute avec une requête utilisateur qui demande une page ou une ressource dans votre application. Cette requête est d'abord traitée par le dispatcheur, qui va sélectionner l'objet controller correct traitant la requête.

Une fois que la requête arrive au controller, celui-ci va communiquer avec la couche Model pour traiter la récupération de données ou les opérations de sauvegarde qui seraient nécessaires. Après que cette communication est finie, le controller va donner à l'objet vue correct, la tâche de générer une sortie résultant des données fournies par le model.

Finalement, quand cette sortie est générée, elle est immédiatement rendu à l'utilisateur.

Presque chaque requête de votre application va suivre ce schéma classique. Nous ajouterons des détails plus tard qui sont spécifiques à CakePHP, donc gardez cela à l'esprit pour la suite.

Bénéfices

Pourquoi utiliser MVC ? Parce que c'est un logiciel vraiment construit selon le patron MVC, qui transforme une application en un dossier élaboré maintenable, modulable et rapide. Elaborer les tâches de l'application en séparant les models, vues et controllers, allègent votre application. De nouvelles fonctionnalités sont ajoutées facilement, et les améliorations sur les vieilles fonctionnalités se font en un clin d'oeil. La conception modulable et séparée permet aussi aux développeurs et designeurs de travailler simultanément, avec la possibilité de prototyper (http ://en.wikipedia.org/wiki/Software_prototyping) rapidement : La séparation permet aussi aux développeurs de faire des changements dans une seule partie de l'application sans affecter les autres.

Si vous n'avez jamais construit une application de cette manière, cela prend quelques temps pour s'habituer, mais nous sommes confiants qu'une fois votre première application construite avec CakePHP, vous ne voudrez plus faire d'une autre façon.

Pour commencer votre première application CakePHP, *Essayez le tutoriel du Blog maintenant*

Où obtenir de l'aide

Le site officiel de CakePHP

http ://www.cakephp.org

Le site officiel de CakePHP est toujours un endroit épatant à visiter. Il propose des liens vers des outils fréquemment utilisés par le développeur, des didacticiels vidéo, des possibilités de faire un don et des téléchargements.

Le Cookbook

http ://book.cakephp.org

Ce manuel devrait probablement être le premier endroit où vous rendre pour obtenir des réponses. Comme pour beaucoup d'autres projets open source, nous accueillons de nouvelles personnes régulièrement. Faîtes tout votre possible pour répondre à vos questions par vous même dans un premier temps. Les réponses peuvent venir lentement, mais elles resteront longtemps et vous aurez ainsi allégé notre charge de travail en support utilisateur. Le manuel et l'API ont tous deux une version en ligne.

La Boulangerie

http ://bakery.cakephp.org

La Boulangerie (Bakery) est une chambre d'enregistrement pour tout ce qui concerne CakePHP. Vous y trouverez des tutoriels, des études de cas et des exemples de code. Lorsque vous serez familiarisés avec CakePHP, connectez-vous pour partager vos connaissances avec la communauté et obtenez en un instant la gloire et la fortune.

L'API

http ://api20.cakephp.org/

Allez droit au but et atteignez le graal des développeurs, l'API CakePHP (Application Programming Interface) est la documentation la plus complète sur tous les détails essentiels au fonctionnement interne du framework. C'est une référence directe au code, donc apportez votre chapeau à hélice.

Les cas de Test

Si vous avez toujours le sentiment que l'information fournie par l'API est insuffisante, regardez le code des cas de test fournis avec CakePHP. Ils peuvent servir d'exemples pratiques pour l'utilisation d'une fonction et de donnée membres d'une classe. :

```
lib/Cake/Test/Case
```

Le canal IRC

Canaux IRC sur irc.freenode.net :
– #cakephp – Discussion générale
– #cakephp-docs – Documentation
– #cakephp-bakery – Bakery
– #cakephp-fr – Canal francophone
Si vous êtes déconcerté, poussez un hurlement sur le canal IRC de CakePHP. Une personne de l'équipe de développement (https ://github.com/cakephp ?tab=members) s'y trouve habituellement, en particulier durant les heures du jour pour les utilisateurs d'Amérique du Nord et du Sud. Nous serions ravis de vous écouter, que vous ayez besoin d'un peu d'aide, que vous vouliez trouver des utilisateurs dans votre région ou que vous aimeriez donner votre nouvelle marque de voiture sportive.

Le Groupe Google

http ://groups.google.com/group/cake-php

CakePHP dispose également d'un Groupe Google très actif. Il peut être une ressource de choix pour trouver des réponses archivées, des questions fréquemment posées et obtenir des réponses aux problèmes urgents.

Les Questions CakePHP

http ://ask.cakephp.org/

Simplement s'inscrire/se connecter et poser une question. Attendez jusqu'à ce que vous ayez quelques réponses et choisissez la réponse correcte. Vous pouvez aussi voir, commenter et voter pour les questions précédemment demandées et aussi les questions résolues.

CHAPITRE 4

Controllers (Contrôleurs)

Les Controllers sont le 'C' dans MVC. Après que le routage a été appliqué et que le bon controller a été trouvé, l'action de votre controller est appelé. Votre controller devra gérer l'interpretation des données requetées, s'assurer que les bon models sont appelés, et que la bonne réponse ou vue est rendue. Les controllers peuvent être imaginés comme un homme au milieu entre le Model et la Vue. Le mieux est de garder des controllers peu chargés, et des models plus fournis. Cela vous aidera à réutiliser plus facilement votre code et facilitera le test de votre code.

Habituellement, les controllers sont utilisés pour gérer la logique autour d'un seul model. Par exemple, si vous construisez un site pour gérer une boulangerie en-ligne, vous aurez sans doute un RecettesController et un IngredientsController qui gérent les recettes et leurs ingrédients. Dans CakePHP, les controllers sont nommés d'après le model principal qu'ils gèrent. Il est aussi totalement possible d'avoir des controllers qui travaillent avec plus d'un model.

Les controllers de votre application sont des classes qui étendent la classe CakePHP `AppController`, qui hérite elle-même de la classe `Controller` du cœur. La classe AppController peut être définie dans `/app/Controller/AppController.php` et elle devrait contenir les méthodes partagées par tous les controllers de votre application.

Les controllers peuvent inclure un certain nombre de méthodes qui sont généralement appelées *actions*. Les actions sont les méthodes d'un controller qui gère les requêtes. Par défaut, toutes les méthodes publiques d'un controller sont des actions accessibles d'une url. Les actions sont responsables de l'interpretation des requêtes et de la création de la réponse. Habituellement, les réponses sont sous forme de vue rendue, mais il y a aussi d'autres façons de créer des réponses.

Le Controller App

Comme indiqué dans l'introduction, la classe AppController est la classe mère de tous les controllers de votre application. AppController étend elle-même la classe Controller incluse dans la librairie du cœur de CakePHP. Ainsi, AppController est définie dans `/app/Controller/AppController.php` comme ceci :

```
class AppController extends Controller {
}
```

Les attributs et méthodes de controller créés dans AppController seront disponibles dans tous les controllers de votre application. C'est l'endroit idéal pour créer du code commun à tous vos controllers. Les Components (que vous découvrirez plus loin) sont mieux appropriés pour du code utilisé dans la plupart (mais pas nécessairement tous) des controllers.

Bien que les règles habituelles d'héritage de la programmation orientée objet soient appliquées, CakePHP exécute également un travail supplémentaire si des attributs spécifiques des controllers sont fournis, comme la liste des components ou helpers utilisés par un controller. Dans ces situations, les valeurs des tableaux de AppController sont fusionnées avec les tableaux de la classe controller enfant. Les valeurs dans la classe enfant vont toujours surcharger celles dans AppController.

Note : CakePHP fusionne les variables suivantes de la classe AppController avec celles des controllers de votre application :
- $components
- $helpers
- $uses

N'oubliez pas d'ajouter les helpers Html et Form si vous avez défini la var `$helpers` dans votre classe AppController.

Pensez à appeler les fonctions de rappel (callbacks) de AppController dans celles du controller enfant pour de meilleurs résultats :

```
public function beforeFilter() {
    parent::beforeFilter();
}
```

Les paramètres de requête

Quand une requête est faîte dans une application CakePHP, Les classes `Router` et `Dispatcher` de CakePHP utilisent la *Configuration des Routes* pour trouver et créer le bon controller. La requête de données est encapsulée dans un objet request. CakePHP met toutes les informations importantes de la requête dans la propriété `$this->request`. Regardez la section *CakeRequest* pour plus d'informations sur l'objet request de CakePHP.

Les Actions du Controller

Les actions du Controller sont responsables de la conversion des paramètres de la requête dans une réponse pour le navigateur/utilisateur faisant la requête. CakePHP utilise les conventions pour automatiser le processus et retirer quelques codes boiler-plate que vous auriez besoin d'écrire autrement.

Par convention, CakePHP rend une vue avec une version inflectée du nom de l'action. Revenons à notre boulangerie en-ligne par exemple, notre RecipesController pourrait con-

tenir les actions `view()`, `share()`, et `search()`. Le controller serait trouvé dans `/app/Controller/RecettesController.php` et contiendrait :

```
# /app/Controller/RecettesController.php

class RecettesController extends AppController {
    public function view($id) {
        //la logique de l'action va ici..
    }

    public function share($client_id, $recette_id) {
        //la logique de l'action va ici..
    }

    public function search($query) {
        //la logique de l'action va ici..
    }
}
```

Les fichiers de vue pour ces actions seraient `app/View/Recettes/view.ctp`, `app/View/Recettes/share.ctp`, et `app/View/Recettes/search.ctp`. Le nom du fichier de vue est par convention le nom de l'action en minuscules et avec des underscores.

Les actions du Controller utilisent généralement `set()` pour créer un contexte que `View` utilise pour rendre la vue. Du fait des conventions que CakePHP utilise, vous n'avez pas à créer et rendre la vue manuellement. Au lieu de ça, une fois qu'une action du controller est terminée, CakePHP va gérer le rendu et la livraison de la Vue.

Si pour certaines raisons, vous voulez échapper au comportement par défaut. Les deux techniques suivantes vont échapper le comportement de rendu par défaut de la vue.

- Si vous retournez une chaîne de caractères, ou un objet qui peut être converti en une chaîne de caractères à partir d'une action du controller, elle sera utilisée comme contenu de réponse.
- Vous pouvez retourner un objet `CakeResponse` avec la réponse complètement créée.

Quand les méthodes du controller sont utilisées avec `requestAction()`, vous voudrez souvent retourner les données qui ne sont pas des chaînes de caractère. Si vous avez des méthodes du controller qui sont utilisées pour des requêtes web normales + requestAction vous devrez vérifier le type de requête avant de retourner :

```
class RecipesController extends AppController {
    public function popular() {
        $popular = $this->Recipe->popular();
        if (!empty($this->request->params['requested'])) {
            return $popular;
        }
        $this->set('popular', $popular);
    }
}
```

Le controller ci-dessus est un exemple montrant comment la méthode peut être utilisée avec `requestAction()` et des requêtes normales. Retourner un tableau de données à une requête non-requestAction va entraîner des erreurs et devra être évité. Regardez la section sur `Controller::requestAction()` pour plus d'astuces sur l'utilisation de `requestAction()`.

Afin que vous utilisiez efficacement le controller dans votre propre application, nous couvrons certains des attributs et méthodes du coeur fournis par les controllers de CakePHP.

Request Life-cycle callbacks

class `Controller`

Les controllers de CakePHP sont livrés par défaut avec des méthodes de rappel (ou callback) qui vous pouvez utiliser pour insérer de la logique juste avant ou juste après que les actions du controller soient effectuées :

`Controller::beforeFilter()`
Cette fonction est exécutée avant chaque action du controller. C'est un endroit pratique pour vérifier le statut d'une session ou les permissions d'un utilisateur.

Note : La méthode beforeFilter() sera appelée pour les actions manquantes et les actions de scaffolding.

`Controller::beforeRender()`
Cette méthode est appelée après l'action du controller mais avant que la vue ne soit rendue. Ce callback n'est pas souvent utilisé, mais peut-être nécessaire si vous appellez render() manuellement à la fin d'une action donnée.

`Controller::afterFilter()`
Cette méthode est appelée après chaque action du controller, et après que l'affichage soit terminé. C'est la dernière méthode du controller qui est exécutée.

En plus des callbacks, *Components (Composants)* fournit aussi un ensemble similaire de callbacks.

Les Méthodes du Controller

Pour une liste complète des méthodes de controller avec leurs descriptions, consultez l'API CakePHP http ://api20.cakephp.org/class/controller.

Interactions avec les vues

Les Controllers interagissent avec la vue de plusieurs façons. Premièrement, ils sont capables de passer des données aux vues, en utilisant `set()`. Vous pouvez aussi décider quelle classe de vue utiliser, et quel fichier de vue doit être rendu à partir du controller.

`Controller::set`(*string $var*, *mixed $value*)
La méthode `set()` est la voie principale utilisée pour transmettre des données de votre controller à votre vue. Une fois `set()` utilisée, la variable de votre controller devient accessible par la vue :

```
// Dans un premier temps vous passez les données depuis le controller:

$this->set('couleur', 'rose');
```

```
// Ensuite vous pouvez les utiliser dans la vue de cette manière:
?>

Vous avez sélectionné un glaçage <?php echo $couleur; ?> pour le gâteau.
```

La méthode `set()` peut également prendre un tableau associatif comme premier paramètre. Cela peut souvent être une manière rapide d'affecter en une seule fois un jeu complet d'informations à la vue. Modifié dans la version 1.3 : Les clefs de votre tableau ne seront plus infléchies avant d'être assignées à la vue ('clef_avec_underscore' ne devient plus 'clefAvecUnderscore', etc...).

```
$data = array(
    'couleur' => 'rose',
    'type' => 'sucre',
    'prix_de_base' => 23.95
);

// donne $couleur, $type, et $prix_de_base
// disponible dans la vue:

$this->set($data);
```

L'attribut `$pageTitle` n'existe plus, utilisez `set()` pour définir le titre :

```
$this->set('title_for_layout', 'Ceci est la page titre');
```

`Controller::render`(*string $action*, *string $layout*, *string $file*)

La méthode `render()` est automatiquement appelée à la fin de chaque action exécutée par le controller. Cette méthode exécute toute la logique liée à la présentation (en utilisant les variables transmises via la méthode `set()`), place le contenu de la vue à l'intérieur de sa mise en page et transmet le tout à l'utilisateur final.

Le fichier de vue utilisé par défaut est déterminé par convention. Ainsi, si l'action `search()` de notre controller RecettesController est demandée, le fichier de vue situé dans /app/view/recettes/search.ctp sera utilisé :

```
class RecettesController extends AppController {
// ...
    public function search() {
        // Rend la vue dans /View/Recettes/search.ctp
        $this->render();
    }
// ...
}
```

Bien que CakePHP appellera cette fonction automatiquement à la fin de chaque action (à moins que vous n'ayez défini `$this->autoRender` à false), vous pouvez l'utiliser pour spécifier un fichier de vue alternatif en précisant le nom d'une action dans le controller, via le paramètre `$action`.

Si `$action` commence avec un '/' on suppose que c'est un fichier de vue ou un élément dont le chemin est relatif au dossier `/app/View`. Cela permet un affichage direct des éléments, ce qui est très pratique lors d'appels ajax.

```
// Rend un élément dans /View/Elements/ajaxreturn.ctp
$this->render('/Elements/ajaxreturn');
```

Vous pouvez aussi spécifier une vue alternative ou un fichier element en utilisant le troisième paramètre, `$file`. Le paramètre `$layout` vous permet de spécifier le layout de la vue qui est rendue.

Rendre une vue spécifique

Dans votre controller, vous pourriez avoir envie de rendre une vue différente que celle rendue par défaut. Vous pouvez faire cela en appelant directement `render()`. Une fois que vous avez appelé `render()` CakePHP n'essaiera pas de re-rendre la vue :

```
class PostsController extends AppController {
    public function mon_action() {
        $this->render('fichier_personnalise');
    }
}
```

Cela rendrait `app/View/Posts/fichier_personnalise.ctp` au lieu de `app/View/Posts/mon_action.ctp`.

Vous pouvez aussi rendre les vues des plugins en utilisant la syntaxe suivante : `$this->render('PluginName.PluginController/custom_file')`. Par exemple :

```
class PostsController extends AppController {
    public function my_action() {
        $this->render('Users.UserDetails/custom_file');
    }
}
```

Cela rendrait la vue `app/Plugin/Users/View/UserDetails/custom_file.ctp`

Contrôle de Flux

`Controller::redirect` (*mixed $url*, *integer $status*, *boolean $exit*)

La méthode de contrôle de flux que vous utiliserez le plus souvent est `redirect()`. Cette méthode prend son premier paramètre sous la forme d'une URL relative à votre application CakePHP. Quand un utilisateur a réalisé un paiement avec succès, vous aimeriez le rediriger vers un écran affichant le reçu. :

```
    public function regler_achats() {
        // Placez ici la logique pour finaliser l'achat...
        if ($success) {
            $this->redirect(array('controller' => 'paiements', 'action' => 'remerciements'
        } else {
            $this->redirect(array('controller' => 'paiements', 'action' => 'confirmations'
        }
    }
```

Vous pouvez aussi utiliser une URL relative ou absolue avec $url :

```
    $this->redirect('/paiements/remerciements'));
    $this->redirect('http://www.exemple.com');
```

Vous pouvez aussi passer des données à l'action :

```
$this->redirect(array('action' => 'editer', $id));
```

Le second paramètre de la fonction `redirect()` vous permet de définir un code de statut HTTP accompagnant la redirection. Vous aurez peut-être besoin d'utiliser le code 301 (document déplacé de façon permanente) ou 303 (voir ailleurs), en fonction de la nature de la redirection.

Cette méthode réalise un `exit()` après la redirection, tant que vous ne mettez pas le troisième paramètre à false.

Si vous avez besoin de rediriger à la page appelante, vous pouvez utiliser :

```
$this->redirect($this->referer());
```

Cette méthode supporte aussi les paramètres nommés de base. Si vous souhaitez être redirigé sur une URL comme : `http://www.example.com/commandes/confirmation/produit:pizza/quantite:` vous pouvez utiliser :

```
$this->redirect(array('controller' => 'commandes', 'action' => 'confirmation', 'produi
```

Un example d'utilisation des requêtes en chaînes et hashés ressemblerait à ceci :

```
$this->redirect(array(
    'controller' => 'commandes', 'action' => 'confirmation', '?' => array('produit' =>
```

L'url généré serait : `http://www.example.com/commandes/confirmation?produit=pizza&quant`

`Controller::flash`(*string $message*, *string $url*, *integer $pause*, *string $layout*)

Tout comme `redirect()`, la méthode `flash()` est utilisée pour rediriger un utilisateur vers une autre page à la fin d'une opération. La méthode `flash()` est toutefois différente en ce sens qu'elle affiche un message avant de diriger l'utilisateur vers une autre url.

Le premier paramètre devrait contenir le message qui sera affiché et le second paramètre une URL relative à votre application CakePHP. CakePHP affichera le `$message` pendant `$pause` secondes avant de rediriger l'utilisateur.

Si vous souhaitez utiliser un template particulier pour messages flash, vous pouvezspécifier le nom du layout dans le paramètre `$layout`.

Pour définir des messages flash dans une page, regardez du côté de la méthode setFlash() du composant Session (SessionComponent).

Callbacks

En plus des *Request Life-cycle callbacks*, CakePHP supporte aussi les callbacks liés au scaffolding.

`Controller::beforeScaffold`(*$method*)

$method nom de la méthode appelée, par exemple index, edit, etc.

`Controller::afterScaffoldSave`(*$method*)

$method nom de la méthode appelée soit edit soit update.

`Controller::afterScaffoldSaveError`(*$method*)

$method nom de la méthode appelée soit edit soit update.

`Controller::scaffoldError`(*$method*)

$method nom de la méthode appelée , par exemple index, edit, etc...

Autres Méthodes utiles

`Controller::constructClasses()`

Cette méthode charge en mémoire les models requis par le controller. Cette procédure de chargement est normalement effectuée par CakePHP, mais cette méthode est à garder sous le coude quand vous avez besoin d'accéder à certains controllers depuis une perspective différente. Si vous avez besoin de CakePHP dans un script utilisable en ligne de commande ou d'autres utilisations externes, constructClasses() peut devenir pratique.

`Controller::referer` (*mixed $default = null*, *boolean $local = false*)

Retourne l'URL référente de la requête courante. Le Paramètre `$default` peut être utilisé pour fournir une URL par défaut à utiliser si HTTP_REFERER ne peut pas être lu par les headers. Donc, au lieu de faire ceci :

```
class UtilisateursController extends AppController {
    public function delete($id) {
        // le code de suppression va ici, et ensuite...
        if ($this->referer() != '/') {
            $this->redirect($this->referer());
        } else {
            $this->redirect(array('action' => 'index'));
        }
    }
}
```

vous pouvez faire ceci :

```
class UtilisateursController extends AppController {
    public function delete($id) {
        // le code de suppression va ici, et ensuite...
        $this->redirect($this->referer(array('action' => 'index')));
    }
}
```

Si `$default` n'est pas défini, la fonction se met par défaut sur à la racine (root) de votre domaine - '/'.

Le paramètre `$local` si il est défini à `true`, restreint les URLs se référant au serveur local.

`Controller::disableCache()`

Utilisée pour indiquer au **navigateur** de l'utilisateur de ne pas mettre en cache le résultat de la requête courante. Ceci est différent du système de cache de vue couvert dans le chapitre suivant.

Les en-têtes HTTP envoyés à cet effet sont :

```
Expires: Mon, 26 Jul 1997 05:00:00 GMT
Last-Modified: [current datetime] GMT
Cache-Control: no-store, no-cache, must-revalidate
Cache-Control: post-check=0, pre-check=0
Pragma: no-cache
```

`Controller::postConditions` (*array $data*, *mixed $op*, *string $bool*, *boolean $exclusive*)

Utilisez cette méthode pour transformer des données de formulaire, transmises par POST (depuis les inputs du Helper Form), en des conditions de recherche pour un model. Cette fonction offre un raccourci appréciable pour la construction de la logique de recherche. Par exemple, un administrateur

aimerait pouvoir chercher des commandes dans le but de connaître quels produits doivent être emballés. Vous pouvez utiliser les Helpers Form et Html pour construire un formulaire rapide basé sur le model Commande. Ensuite une action du controller peut utiliser les données postées par ce formulaire pour construire automatiquement les conditions de la recherche :

```
public function index() {
    $conditions = $this->postConditions($this->request->data);
    $commandes = $this->Commande->find('all', compact('conditions'));
    $this->set('commandes', $orders);
}
```

Si `$this->data['Commande']['destination']` vaut "Boulangerie du village", postConditions convertit cette condition en un tableau compatible avec la méthode Model->find(). Soit dans notre cas, `array("Commande.destination" => "Boulangerie du village")`.

Si vous voulez utiliser un opérateur SQL différent entre chaque terme, remplacez-le en utilisant le second paramètre :

```
/*
Contenu de $this->request->data
array(
    'Commande' => array(
        'nb_items' => '4',
        'referrer' => 'Ye Olde'
    )
)
*/

// Récupérons maintenant les commandes qui ont au moins 4 items et
contenant 'Ye Olde'
$conditions = $this->postConditions(
    $this->request->data,
    array(
        'nb_items' => '>=',
        'referrer' => 'LIKE'
    )
);
$commandes = $this->Commande->find('all', compact('conditions'));
```

Le troisième paramètre vous permet de dire à CakePHP quel opérateur booléen SQL utiliser entre les conditions de recherche. Les chaînes comme 'AND', 'OR' et 'XOR' sont des valeurs possibles.

Enfin, si le dernier paramètre est défini à vrai et que $op est un tableau, les champs non-inclus dans $op ne seront pas inclus dans les conditions retournées.

`Controller::paginate()`

Cette méthode est utilisée pour paginer les résultats retournés par vos models. Vous pouvez définir les tailles de la page, les conditions à utiliser pour la recherche de ces données et bien plus. Consultez la section *pagination* pour plus de détails sur l'utilisation de la pagination.

`Controller::requestAction` (*string $url*, *array $options*)

Cette fonction appelle l'action d'un controller depuis tout endroit du code et retourne les données associées à cette action. L'`$url` passée est une adresse relative à votre application CakePHP (/nomducontroleur/nomaction/parametres). Pour passer des données supplémentaires au controller destinataire ajoutez le tableau $options.

Note : Vous pouvez utiliser `requestAction()` pour récupérer l'intégralité de l'affichage d'une vue en passant la valeur 'return' dans les options : `requestAction($url, array('return'))`. Il est important de noter que faire un requestAction en utilisant 'return' à partir d'une méthode d'un controller peut entraîner des problèmes de fonctionnement dans les script et tags css.

Warning : Si elle est utilisée sans cache, la méthode `requestAction` peut engendrer des faibles performances. Il est rarement approprié de l'utiliser dans un controller ou un model.

`requestAction` est plutôt utilisé en conjonction avec des éléments (mis en cache) - comme moyen de récupérer les données pour un élément avant de l'afficher. Prenons l'exemple de la mise en place d'un élément "derniers commentaires" dans le gabarit (layout). Nous devons d'abord créer une méthode de controller qui retourne les données :

```
// Controller/CommentsController.php
class CommentsController extends AppController {
    public function latest() {
        if (empty($this->request->params['requested'])) {
            throw new ForbiddenException();
        }
        return $this->Comment->find('all', array('order' => 'Comment.created DESC', 'l
    }
}
```

Vous devriez toujours inclure des vérifications pour vous assurer que vos méthodes de requestAction sont en fait originaires de `requestAction`. Ne pas le faire va autoriser les méthodes requestAction à être directement accessible d'une URL, ce qui est généralement non souhaité.

Si nous créons un élément simple pour appeler cette fonction :

```
// View/Elements/latest_comments.ctp

$comments = $this->requestAction('/comments/latest');
foreach ($comments as $comment) {
    echo $comment['Comment']['title'];
}
```

On peut ensuite placer cet élément n'importe où pour obtenir la sortie en utilisant :

```
echo $this->element('latest_comments');
```

Ecrit de cette manière, dès que l'élément est affiché, une requête sera faite au controller pour obtenir les données, les données seront traitées, et retournées. Cependant, compte tenu de l'avertissement ci-dessus il vaut mieux utiliser des éléments mis en cache pour anticiper des traitements inutiles. En modifiant l'appel à l'élément pour qu'il ressemble à ceci :

```
echo $this->element('latest_comments', array(), array('cache' => true));
```

L'appel à `requestAction` ne sera pas effectué tant que le fichier de vue de l'élément en cache existe et est valide.

De plus, `requestAction` prend désormais des urls basées sur des tableau dans le style de cake :

```
echo $this->requestAction(
    array('controller' => 'articles', 'action' => 'featured'),
    array('return')
);
```

Cela permet à l'appel de requestAction d'éviter l'utilisation de Router : :url ce qui peut améliorer la performance. Les url basées sur des tableaux sont les mêmes que celles utilisées par `HtmlHelper::link()` avec une seule différence. Si vous utilisez des paramètres nommés ou passés dans vos url, vous devez les mettre dans un second tableau et les inclures dans la clé correcte. La raison de cela est que requestAction fusionne seulement le tableau des arguments nommés avec les membres du tableau de Controller : :params et ne place pas les arguments nommés dans la clé 'named'. Des membres supplémentaires dans le tableau `$option` va aussi être rendu disponible dans le tableau Controller : :params de l'action requêtée

```
echo $this->requestAction('/articles/featured/limit:3');
echo $this->requestAction('/articles/view/5');
```

En array dans requestAction serait ainsi :

```
echo $this->requestAction(
    array('controller' => 'articles', 'action' => 'featured'),
    array('named' => array('limit' => 3))
);

echo $this->requestAction(
    array('controller' => 'articles', 'action' => 'view'),
    array('pass' => array(5))
);
```

Note : Contrairement aux autres places où les urls en tableau sont analogues aux urls en chaîne de caractère, requestAction les traite différemment.

Quand vous utilisez une url en tableau en conjonction avec requestAction(), vous devez spécifier **tous** les paramètres dont vous aurez besoin dans l'action requêtée. Ceci inclut les paramètres comme `$this->request->data`. En plus de passer tous les paramètres requis, les paramètres nommés et passés doivent être faits dans le second tableau comme vu ci-dessus.

`Controller::`**`loadModel`**(*string $modelClass*, *mixed $id*)

La fonction `loadModel` devient pratique quand vous avez besoin d'utiliser un model qui n'est pas le model du controller par défaut ou un de ses models associés :

```
$this->loadModel('Article');
$recentArticles = $this->Article->find('all', array('limit' => 5, 'order' => 'Article.

$this->loadModel('User', 2);
$user = $this->User->read();
```

Les attributs du Controller

Pour une liste complète des attributs du controller et ses descriptions, visitez l'API de CakePHP. Regardez http ://api20.cakephp.org/class/controller.

property `Controller::$name`

L'attribut `$name` doit être défini selon le nom du controller. Habituellement, c'est juste la forme plurielle du model principal que le controller utilise. Cette propriété n'est pas requis, mais évite à CakePHP d'inflecter dessus :

```
// Exemple d'utilisation d'attribut $name du controller
class RecipesController extends AppController {
   public $name = 'Recipes';
}
```

$components, $helpers et $uses

Les autres attributs les plus souvent utilisés permettent d'indiquer à CakePHP quels helpers, components et models vous utiliserez avec le controller courant. Utiliser ces attributs rend ces classes MVC, fournies par `$components` et `$uses`, disponibles pour le controller, sous la forme de variables de classe (`$this->ModelName`, par exemple) et celles fournies par `$helpers`, disponibles pour la vue comme une variable référence à l'objet (`$this->{$helpername}`).

Note : Chaque controller a déjà accès, par défaut, à certaines de ces classes, donc vous n'avez pas besoin de les redéfinir.

---

property `Controller::$uses`

Les controllers ont accès par défaut à leur model primaire respectif. Notre controller Recettes aura donc accès à son model Recette, disponible via `$this->Recette`, et notre controller Produits proposera un accès à son model via `$this->Produit`. Cependant, quand vous autorisez un controller à accéder à d'autres models via la variable `$uses`, le nom du model primaire du controller courant doit également être inclu. Ceci est illustré dans l'exemple ci-dessous.

Si vous ne souhaitez pas utilisez un Model dans votre controller, définissez `public $uses = array()`. Cela vous permettra d'utiliser un controller sans avoir besoin d'un fichier Model correspondant. Cependant, les models définis dans `AppController` seront toujours chargés. Vous pouvez aussi utiliser `false` pour ne charger absolument aucun model. Même ceux définis dans `AppController`. Modifié dans la version 2.1 : Uses a maintenant une nouvelle valeur par défaut, il gère aussi `false` différemment.

property `Controller::$helpers`

Les Helpers Html et Session sont toujours accessibles par défaut, tout comme le SessionComponent. Mais si vous choisissez de définir votre propre tableau `$helpers` dans AppController, assurez-vous d'y inclure `Html` et `Form` si vous voulez qu'ils soient toujours disponibles par défaut dans vos propres controllers. Pour en savoir plus au sujet de ces classes, assurez-vous de regarder leurs sections respectives plus loin dans le manuel.

Jetons maintenant un œil sur la façon d'indiquer à un controller CakePHP que vous avez dans l'idée d'utiliser d'autres classes MVC :

```
class RecipesController extends AppController {
    public $uses = array('Recipe', 'User');
    public $helpers = array('Js');
    public $components = array('RequestHandler');
}
```

Toutes ces variables sont fusionnées avec leurs valeurs héritées, par conséquent ce n'est pas nécessaire de re-déclarer (par exemple) le helper Form ou tout autre déclaré dans votre controller App.

property `Controller::$components`

Le tableau de components vous permet de définir quel *Components (Composants)* un controller va utiliser. Comme les `$helpers` et `$uses`, les components dans vos controllers sont fusionnés avec ceux dans `AppController`. Comme pour les `$helpers`, vous pouvez passer les paramètres dans les components. Regardez *Configuration des Components* pour plus d'informations.

Autres Attributs

Tandis que vous pouvez vérifier les détails pour tous les attributs des controllers dans l'API, il y a d'autres attributs du controller qui méritent leurs propres sections dans le manuel.

En savoir plus sur les controllers

Les Objets Request et Response

Les objets request et response sont nouveaux depuis CakePHP 2.0. Dans les versions précédentes, ces objets étaient représentés à travers des tableaux, et les méthodes liées étaient utilisées à travers `RequestHandlerComponent`, `Router`, `Dispatcher` et `Controller`. Il n'y avait pas d'objet global qui reprenait les informations de la requête. Depuis CakePHP 2.0, `CakeRequest` et `CakeResponse` sont utilisés pour cela.

CakeRequest

`CakeRequest` est l'objet requête utilisé par défaut dans CakePHP. Il centralise un certain nombre de fonctionnalités pour interroger et intéragir avec les données demandées. Pour chaque requête, une CakeRequest est créée et passée en référence aux différentes couches de l'application que la requête de données utilise. Par défaut `CakeRequest` est assignée à `$this->request`, et est disponible dans les Controllers, Vues et Helpers. Vous pouvez aussi y accéder dans les Components en utilisant la référence du controller. Certaines des tâches incluses que `CakeRequest` permet :

- Transformer les tableaux GET, POST, et FILES en structures de données avec lesquelles vous êtes familiers.
- Fournir une introspection de l'environnement se rapportant à la demande. Des choses comme les envois d'en-têtes (headers), l'adresse IP du client et les informations des sous-domaines/domaines sur lesquels le serveur de l'application tourne.
- Fournit un accès aux paramètres de la requête à la fois en tableaux indicés et en propriétés d'un objet.

Accéder aux paramètres de la requête

CakeRequest propose plusieurs interfaces pour accéder aux paramètres de la requête. La première est des tableaux indexés, la seconde est à travers `$this->request->params`, et la troisième est des propriétés d'objets :

```
$this->request['controller'];
$this->request->controller;
$this->request->params['controller']
```

Tout ce qui est au-dessus retournera la même valeur. Plusieurs façons d'accéder aux paramètres a été fait pour faciliter la migration des applications existantes. Tous les éléments de route *Les éléments de Route* sont accessibles à travers cette interface.

En plus des éléments de routes *Les éléments de Route*, vous avez souvent besoin d'accéder aux arguments passés *Arguments passés* et aux paramètres nommés *Paramètres nommés*. Ceux-ci sont aussi tous les deux disponibles dans l'objet request :

```
// Arguments passés
$this->request['pass'];
$this->request->pass;
$this->request->params['pass'];

// Paramètres nommés
$this->request['named'];
$this->request->named;
$this->request->params['named'];
```

Il vous fournira un accès aux arguments passés et ax paramètres nommés. Il y a de nombreux paramètres importants et utiles que CakePHP utilise en interne, il sont aussi trouvables dans les paramètres de la requête :

- `plugin` Le plugin gérant la requête, va être nul pour les non-plugins.
- `controller` Le controller gère la requête courante.
- `action` L'action gère la requête courante.
- `prefix` Le prefixe pour l'action courante. Voir *Prefix de routage* pour plus d'informations.
- `bare` Présent quand la requête vient de requestAction() et inclut l'option bare. Les requêtes vides n'ont pas de layout de rendu.
- `requested` Présent et mis à true quand l'action vient de requestAction.

Accéder aux paramètres Querystring

Les paramètres Querystring peuvent être lus en utilisant `CakeRequest::$query` :

```
// l\'url est /posts/index?page=1&sort=title
$this->request->query['page'];

//  Vous pouvez aussi y accéder par un tableau
$this->request['url']['page'];
```

Accéder aux données POST

Toutes les données POST peuvent être atteintes à travers `CakeRequest::$data`. N'importe quelle forme de tableau qui contient un prefixe `data`, va avoir sa donnée prefixée retirée. Par exemple :

```
// Un input avec un nom attribute égal à 'data[MyModel][title]' est accessible
$this->request->data['MyModel']['title'];
```

Vous pouvez soit accéder directement à la propriété des données, soit vous pouvez utiliser `CakeRequest::data()` pour lire le tableau de données sans erreurs. N'importe quelle clé qui n'existe pas va retourner `null` :

```
$foo = $this->request->data('Value.that.does.not.exist');
// $foo == null
```

Accéder aux données PUT ou POST

Introduit dans la version 2.2. Quand vous construisez des services REST, vous acceptez souvent des données requêtées sur des requêtes `PUT` et `DELETE`. Depuis 2.2, toute donnée de corps de requête `application/x-www-form-urlencoded` va automatiquement être parsée et définie dans `$this->data` pour les requêtes `PUT` et `DELETE`. Si vous acceptez les données JSON ou XML, regardez ci-dessous comment vous pouvez accéder aux corps de ces requêtes.

Accéder aux données XML ou JSON

Les applications employant *REST* échangent souvent des données dans des organes post non encodées en URL. Vous pouvez lire les données entrantes dans n'importe quel format en utilisant `CakeRequest::input()`. En fournissant une fonction de décodage, vous pouvez recevoir le contenu dans un format déserializé :

```
// Obtenir les données encodées JSON soumises par une action PUT/POST
$data = $this->request->input('json_decode');
```

Puisque certaines méthodes de desérialization ont besoin de paramètres supplémentaires quand elles sont appelées, comme le paramètre de type tableau pour `json_decode` ou si vous voulez convertir les XML en objet DOMDocument, `CakeRequest::input()` supporte aussi le passement dans des paramètres supplémentaires :

```
// Obtenir les données encodées en Xml soumises avec une action PUT/POST
$data = $this->request->input('Xml::build', array('return' => 'domdocument'));
```

Accéder aux informations du chemin

CakeRequest fournit aussi des informations utiles sur les chemins dans votre application. `CakeRequest::$base` et `CakeRequest::$webroot` sont utiles pour générer des urls, et déterminer si votre application est ou n'est pas dans un sous-dossier.

Inspecter la requête

Détecter les différentes conditions de la requête utilisée en utilisant `RequestHandlerComponent`. Ces méthodes ont été déplacées dans `CakeRequest`, et offrent une nouvelle interface compatible avec les utilisations anciennes :

```
$this->request->is('post');
$this->request->isPost();
```

Les deux méthodes appelées vont retourner la même valeur. Pour l'instant, les méthodes sont toujours disponibles dans RequestHandler, mais sont depréciées et pourraient être retirées avant la version finale. Vous pouvez aussi facilement étendre les détecteurs de la requête qui sont disponibles, en utilisant `CakeRequest::addDetector()` pour créer de nouveaux types de détecteurs. Il y a quatre différents types de détecteurs que vous pouvez créer :

- Comparaison avec valeur d'environnement - Une comparaison de la valeur d'environnement, compare une valeur attrapée à partir de `env()` pour une valeur connue, la valeur d'environnement est vérifiée équitablement avec la valeur fournie.
- La comparaison de la valeur model - La comparaison de la valeur model vous autorise à comparer une valeur attrapée à partir de `env()` à une expression régulière.
- Comparaison basée sur les options - La comparaison basée sur les options utilise une liste d'options pour créer une expression régulière. De tels appels pour ajouter un détecteur d'options déjà défini, va fusionner les options.
- Les détecteurs de Callback - Les détecteurs de Callback vous permettront de fournir un type 'callback' pour gérer une vérification. Le callback va recevoir l'objet requête comme seul paramètre.

Quelques exemples seraient :

```
// Ajouter un détecteur d'environment.
$this->request->addDetector('post', array('env' => 'REQUEST_METHOD', 'value' => 'POST'));

// Ajouter un détecteur de valeur model.
$this->request->addDetector('iphone', array('env' => 'HTTP_USER_AGENT', 'pattern' => '/iPho

// Ajouter un détecteur d'options
$this->request->addDetector('internalIp', array(
    'env' => 'CLIENT_IP',
    'options' => array('192.168.0.101', '192.168.0.100')
));

// Ajouter un détecteur de callback. Peut soit être une fonction anonyme ou un callback rég
$this->request->addDetector('awesome', array('callback' => function ($request) {
    return isset($request->awesome);
}));
```

`CakeRequest` inclut aussi des méthodes comme `CakeRequest::domain()`, `CakeRequest::subdomains()` et `CakeRequest::host()` qui aident les applications avec sous-domaines à avoir une vie un peu plus facile.

Il y a des plusieurs détecteurs intégrés que vous pouvez utiliser :

- `is('get')` Vérifier pour voir si la requête courante est un GET.
- `is('put')` Vérifier pour voir si la requête courante est un PUT.
- `is('post')` Vérifier pour voir si la requête courante est un POST.

- `is('delete')` Vérifier pour voir si la requête courante est un DELETE.
- `is('head')` Vérifier pour voir si la requête courante est un HEAD.
- `is('options')` Vérifier pour voir si la requête courante est OPTIONS.
- `is('ajax')` Vérifier pour voir si la requête courante vient d'un X-Requested-with = XmlHttpRequest.
- `is('ssl')` Vérifier pour voir si la requête courante est via SSL.
- `is('flash')` Vérifier pour voir si la requête courante a un User-Agent de Flash.
- `is('mobile')` Vérifier pour voir si la requête courante vient d'une liste courante de mobiles.

CakeRequest et RequestHandlerComponent

Puisque plusieurs des fonctionnalités offertes par `CakeRequest` étaient l'apanage de `RequestHandlerComponent` une reflexion était nécessaire pour savoir si il était toujours nécessaire. Dans 2.0, `RequestHandlerComponent` agit comme un sugar daddy. En fournissant une couche de facilité en haut de l'offre utilitaire de *CakeRequest*. Facilité comme changer les layouts et vues basés sur les types de contenu ou ajax est le domaine de `RequestHandlerComponent`. Cette séparation des utilitaires entre les deux classes vous laisse un choix plus facile de prendre ce dont vous avez besoin.

Interagir avec les autres aspects de la requête

Vous pouvez utiliser *CakeRequest* pour faire une introspection d'une variété de choses au sujet de la requête. Au-delà des détecteurs, vous pouvez également trouver d'autres informations aux propriétés et méthodes diverses.
- `$this->request->webroot` contient le répertoire webroot.
- `$this->request->base` contient le chemin de base.
- `$this->request->here` contient l'adresse complète de la requête courante.
- `$this->request->query` contient les paramètres de la chaîne de requête.

API de CakeRequest

class `CakeRequest`

CakeRequest encapsule la gestion des paramètres de la requête, et son introspection.

`CakeRequest::domain()`

Retourne le nom de domaine sur lequel votre application tourne.

`CakeRequest::subdomains()`

Retourne le sous-domaine sur lequel votre application tourne en tableau.

`CakeRequest::host()`

Retourne l'hôte où votre application tourne.

`CakeRequest::method()`

Retourne la méthode HTTP où la requête a été faite.

`CakeRequest::onlyAllow(`*$methods*`)`

Défini les méthodes HTTP autorisées, si elles ne correspondent pas, elle va lancer une MethodNotAllowedException La réponse 405 va inclure l'en-tête 'Allow' nécessaire avec les méthodes passées. Introduit dans la version 2.3.

`CakeRequest::referer`(*$local = false*)
Retourne l'adresse de référence de la requête.

`CakeRequest::clientIp`(*$safe = true*)
Retourne l'adresse IP du visiteur courant.

`CakeRequest::header`(*$name*)
Vous permet d'accéder à tout en-tête `HTTP_*` utilisé pour la requête :

```
$this->request->header('User-Agent');
```

Retournerait le user agent utilisé pour la requête.

`CakeRequest::input`(*$callback*[, *$options*])
Récupère les données d'entrée pour une requête, et les passe optionnellement à travers une fonction qui décode. Les paramètres supplémentaires pour la fonction décondant peuvent être passés comme des arguments de input().

`CakeRequest::data`(*$name*)
Fournit une notation avec point pour accéder aux données requêtées. Permet la lecture et la modification des données requêtées, les appels peuvent aussi être chaînés ensemble :

```
// Modifier une donnée requêtée, ainsi vous pouvez pré-enregistrer certains champs.
$this->request->data('Post.title', 'New post')
    ->data('Comment.1.author', 'Mark');

// Vous pouvez aussi lire des données.
$value = $this->request->data('Post.title');
```

`CakeRequest::query`(*$name*)
Fournit un accès aux données requêtées de l'url avec notation en point :

```
// l\'url est /posts/index?page=1&sort=title
$value = $this->request->query('page');
```

Introduit dans la version 2.3.

`CakeRequest::is`(*$type*)
Vérifie si la requête remplit certains critères ou non. Utilisez les règles de détection déjà construites ainsi que toute règle supplémentaire définie dans `CakeRequest::addDetector()`.

`CakeRequest::addDetector`(*$name*, *$options*)
Ajoute un détecteur pour être utilisé avec is(). Voir *Inspecter la requête* pour plus d'informations.

`CakeRequest::accepts`(*$type = null*)
Trouve quels types de contenu le client accepte ou vérifie si ils acceptent un type particulier de contenu.

Récupère tous les types :

```
<?php
$this->request->accepts();
```

Vérifie pour un simple type :

```
$this->request->accepts('application/json');
```

static `CakeRequest::acceptLanguage` (*$language = null*)

Obtenir soit toutes les langues acceptées par le client, ou vérifier si une langue spécifique est acceptée.

Obtenir la liste des langues acceptées :

```
CakeRequest::acceptLanguage();
```

Vérifier si une langue spécifique est acceptée :

```
CakeRequest::acceptLanguage('es-es');
```

property `CakeRequest::$data`

Un tableau de données POST. Vous pouvez utiliser `CakeRequest::data()` pour lire cette propriété d'une manière qui supprime les erreurs notice.

property `CakeRequest::$query`

Un tableau des paramètres de chaîne requêtés.

property `CakeRequest::$params`

Un tableau des éléments de route et des paramètres requêtés.

property `CakeRequest::$here`

Retourne la requête uri courante.

property `CakeRequest::$base`

Le chemin de base de l'application, normalement `/` à moins que votre application soit dans un sous-répertoire.

property `CakeRequest::$webroot`

Le webroot courant.

CakeResponse

`CakeResponse` est la classe de réponse par défaut dans CakePHP. Elle encapsule un nombre de fonctionnalités et de caractéristiques pour la génération de réponses HTTP dans votre application. Elle aide aussi à tester puisqu'elle peut être mocked/stubbed, vous permettant d'inspecter les en-têtes qui vont être envoyés. Comme `CakeRequest`, `CakeResponse` consolide un nombre de méthodes qu'on pouvait trouver avant dans `Controller`, `RequestHandlerComponent` et `Dispatcher`. Les anciennes méthodes sont dépréciés en faveur de l'utilisation de `CakeResponse`.

`CakeResponse` fournit une interface pour envelopper les tâches de réponse communes liées, telles que :

- Envoyer des en-têtes pour les redirections.
- Envoyer des en-têtes de type de contenu.
- Envoyer tout en-tête.
- Envoyer le corps de la réponse.

Changer la classe de réponse

CakePHP utilise `CakeResponse` par défaut. `CakeResponse` est flexible et transparente pour l'utilisation de la classe. Mais si vous avez besoin de la remplacer avec une classe spécifique de l'application, vous pouvez l'écraser et remplacer `CakeResponse` avec votre propre classe. En remplaçant la CakeResponse utilisé dans index.php.

Cela fera que tous les controllers dans votre application utiliseront `VotreResponse` au lieu de `CakeResponse`. Vous pouvez aussi remplacer l'instance de réponse utilisée par la configuration `$this->response` dans vos controllers. Ecraser l'objet réponse est à portée de main pour les tests car il vous permet d'écraser les méthodes qui interragissent avec `header()`. Voir la section sur *CakeResponse et les tests* pour plus d'informations.

Gérer les types de contenu

Vous pouvez contrôler le Content-Type des réponses de votre application en utilisant `CakeResponse::type()`. Si votre application a besoin de gérer les types de contenu qui ne sont pas construits dans CakeResponse, vous pouvez mapper ces types avec `type()` comme ceci :

```
// Ajouter un type vCard
$this->response->type(array('vcf' => 'text/v-card'));

// Configurer la réponse de Type de Contenu pour vcard.
$this->response->type('vcf');
```

Habituellement, vous voudrez mapper des types de contenu supplémentaires dans votre callback `beforeFilter` dans votre controller, afin que vous puissiez tirer parti de la fonctionnalité de vue de commutation automatique de `RequestHandlerComponent` si vous l'utilisez.

Envoyer des pièces jointes

Il y a des fois où vous voulez envoyer des réponses du Controller en fichier à télécharger. Avant la version 2.3, vous pouviez utiliser *Vues Media* pour faire cela. Depuis 2.3 MediaView est déprécié et vous pouvez utiliser `CakeResponse::file()` pour envoyer un fichier en réponse :

```
public function sendFile($id) {
    $file = $this->Attachment->getFile($id);
    $this->response->file($file['path']);
    //Retourne un objet reponse pour éviter que le controller n'essaie de rendre la vue
    return $this->response;
}
```

Comme montré dans l'exemple ci-dessus et comme prévu, vous devez passer le chemin du fichier à la méthode. Cake va envoyer le bon en-tête de type de contenu si c'est un type de fichier connu listé dans *CakeReponse : :$_mimeTypes*. Vous pouvez ajouter des nouveaux types avant d'appeler `CakeResponse::file()` en utilisant la méthode `CakeResponse::type()`.

Si vous voulez, vous pouvez aussi forcer un fichier à être téléchargé au lieu d'être affiché dans le navigateur en spécifiant les options :

```
$this->response->file($file['path'], array('download' => true, 'name' => 'foo'));
```

Définir les en-têtes

Le réglage des en-têtes est fait avec la métode `CakeResponse::header()`. Elle peut être appelée avec quelques paramètres de configurations :

```
// Régler un unique en-tête
$this->response->header('Location', 'http://example.com');

// Régler plusieurs en-têtes
$this->response->header(array('Location' => 'http://example.com', 'X-Extra' => 'My header'
$this->response->header(array('WWW-Authenticate: Negotiate', 'Content-type: application/pd
```

Régler le même en-tête de multiples fois entraînera l'écrasement des précédentes valeurs, un peu comme les appels réguliers d'en-tête. Les en-têtes ne sont aussi pas envoyés quand `CakeResponse::header()` est appelé. Ils sont simplement conservés jusqu'à ce que la réponse soit effectivement envoyé. Introduit dans la version 2.4. Vous pouvez maintenant utiliser la méthode pratique `CakeResponse::location()` pour directement définir ou récupérer l'en-tête de localisation du redirect.

Interragir avec le cache du navigateur

Vous avez parfois besoin de forcer les navigateurs à ne pas mettre en cache les résultats de l'action d'un controller. `CakeResponse::disableCache()` est justement prévu pour cela :

```
public function index() {
    // faire quelque chose.
    $this->response->disableCache();
}
```

> **Warning :** En utilisant disableCache() avec downloads à partir de domaines SSL pendant que vous essayez d'envoyer des fichiers à Internet Explorer peut entraîner des erreurs.

Vous pouvez aussi dire aux clients que vous voulez qu'ils mettent en cache des réponses. En utilisant `CakeResponse::cache()` :

```
public function index() {
    //faire quelque chose
    $this->response->cache('-1 minute', '+5 days');
}
```

Ce qui est au-dessus dira aux clients de mettre en cache la réponse résultante pendant 5 jours, en espérant accélerer l'expérience de vos visiteurs. `cache()` défini la valeur dernièrement modifiée en premier argument. Expires, et Max-age sont définis en se basant sur le second paramètre. Le Cache-Control est défini aussi à public.

Réglage fin du Cache HTTP

Une des façons les meilleures et les plus simples de rendre votre application plus rapide est d'utiliser le cache HTTP. Avec la mise en cache des models, vous n'avez qu'à aider les clients à décider si ils devraient utiliser une copie mise en cache de la réponse en configurant un peu les en-têtes comme les temps modifiés, les balise d'entité de réponse et autres.

Opposé à l'idée d'avoir à coder la logique de mise en cache et de sa nullité (rafraîchissement) une fois que les données ont changé, HTPP utilise deux models, l'expiration et la validation qui habituellement sont beaucoup plus simples que d'avoir à gérer le cache soi-même.

En dehors de l'utilisation de `CakeResponse::cache()` vous pouvez aussi utiliser plusieurs autres méthodes pour affiner le réglage des en-têtes de cache HTTP pour tirer profit du navigateur ou à l'inverse du cache du proxy.

L'en-tête de Cache Control

Introduit dans la version 2.1. Utilisé sous le model d'expiration, cet en-tête contient de multiples indicateurs qui peuvent changer la façon dont les navigateurs ou les proxies utilisent le contenu mis en cache. Un en-tête Cache-Control peut ressembler à ceci :

```
Cache-Control: private, max-age=3600, must-revalidate
```

La classe `CakeResponse` vous aide à configurer cet en-tête avec quelques méthodes utiles qui vont produire un en-tête final valide Cache Control. Premièrement il y a la méthode `CakeResponse::sharable()`, qui indique si une réponse peut être considerée comme partageable pour différents utilisateurs ou clients. Cette méthode contrôle généralement la partie *public* ou *private* de cet en-tête. Définir une réponse en privé indique que tout ou une partie de celle-ci est prévue pour un unique utilisateur. Pour tirer profit des mises en cache partagées, il est nécessaire de définir la directive de contrôle en publique.

Le deuxième paramètre de cette méthode est utilisé pour spécifier un *max-age* pour le cache, qui est le nombre de secondes après lesquelles la réponse n'est plus considérée comme récente. :

```
public function view() {
    ...
    // Défini le Cache-Control en public pour 3600 secondes
    $this->response->sharable(true, 3600);
}

public function mes_donnees() {
    ...
    // Défini le Cache-Control en private pour 3600 secondes
    $this->response->sharable(false, 3600);
}
```

`CakeResponse` expose des méthodes séparées pour la définition de chaque component dans l'en-tête de Cache-Control.

L'en-tête d'Expiration

Introduit dans la version 2.1. Aussi sous le model d'expiration de cache, vous pouvez définir l'en-tête *Expires*, qui selon la spécification HTTP est la date/le temps après que la réponse ne soit plus considerée comme récent. Cet en-tête peut être défini en utilisant la méthode `CakeResponse::expires()` :

```
public function view() {
    $this->response->expires('+5 days');
}
```

Cette méthode accepte aussi un DateTime ou toute chaîne de caractère qui peut être parsée par la classe DateTime.

L'en-tête Etag

Introduit dans la version 2.1. Cache validation dans HTTP est souvent utilisé quand le contenu change constamment et demande à l'application de générer seulement les contenus réponse si le cache n'est plus récent. Sous ce model, le client continue de stocker les pages dans le cache, mais au lieu de l'utiliser directement, il demande à l'application à chaque fois si les ressources ont changé ou non. C'est utilisé couramment avec des ressources statiques comme les images et autres choses.

L'en-tête Etag (appelé balise d'entité) est une chaîne de caractère qui identifie de façon unique les ressource requêtées. Il est très semblable à la somme de contrôle d'un fichier, la mise en cache permettra de comparer les sommes de contrôle pour savoir si elles correspondent ou non.

Pour tirer réellement avantage pour l'utilisation de cet en-tête, vous devez soit appeler manuellement la méthode `CakeResponse::checkNotModified()`, soit avoir le `RequestHandlerComponent` inclu dans votre controller :

```
public function index() {
    $articles = $this->Article->find('all');
    $this->response->etag($this->Article->generateHash($articles));
    if ($this->response->checkNotModified($this->request)) {
        return $this->response;
    }
    ...
}
```

L'en-tête Dernier Modifié

Introduit dans la version 2.1. Toujours dans le cadre du model de validation du cache HTTP, vous pouvez définir l'en-tête *Dernier-Modifié* pour indiquer la date et le temps pendant lequel la ressource a été modifiée pour la dernière fois. Définir cet en-tête aide la réponse de CakePHP pour mettre en cache les clients si la réponse a été modifiée ou n'est pas basée sur le cache du client.

Pour tirer réellement avantage pour l'utilisation de cet en-tête, vous devez soit appeler manuellement la méthode `CakeResponse::checkNotModified()`, soit avoir le `RequestHandlerComponent` inclu dans votre controller :

```
public function view() {
    $article = $this->Article->find('first');
    $this->response->modified($article['Article']['modified']);
    if ($this->response->checkNotModified($this->request)) {
        return $this->response;
    }
    ...
}
```

L'en-tête Vary

Dans certains cas, vous voudrez offrir différents contenus en utilisant la même url. C'est souvent le cas quand vous avez une page multilingue ou que vous répondez avec du HTML différent selon le navigateur qui requête la ressource. Pour ces circonstances, vous pouvez utiliser l'en-tête Vary :

```
$this->response->vary('User-Agent');
$this->response->vary('Accept-Encoding', 'User-Agent');
$this->response->vary('Accept-Language');
```

CakeResponse et les tests

Probablement l'une des plus grandes victoires de `CakeResponse` vient de comment il facilite les tests des controllers et des components. Au lieu de méthodes répandues à travers plusieurs objets, vous avez un seul objet pour mocker pendant que les controllers et les components déleguent à `CakeResponse`. Cela vous aide à rester plus près d'un test 'unit' et facilite les tests des controllers :

```
public function testSomething() {
    $this->controller->response = $this->getMock('CakeResponse');
    $this->controller->response->expects($this->once())->method('header');
    // ...
}
```

De plus, vous pouvez faciliter encore plus l'exécution des tests à partir d'une ligne de commande, pendant que vous pouvez mocker pour éviter les erreurs 'd'envois d'en-têtes' qui peuvent arriver en essayant de configurer les en-têtes dans CLI.

API de CakeResponse

class `CakeResponse`

CakeResponse fournit un nombre de méthodes utiles pour interagir avec la réponse que vous envoyez à un client.

`CakeResponse::`**`header`**(*$header = null*, *$value = null*)

Vois permet de configurer directement un ou plusieurs en-têtes à être envoyés avec la réponse.

`CakeResponse::`**`location`**(*$url = null*)

Vous permet de définir directement l'en-tête de localisation du redirect à envoyer avec la réponse.

// Définit la localisation du redirect $this->response->location('http ://example.com') ;

// Récupère l'en-tête de localisation du redirect actuel $location = $this->response->location() ; Introduit dans la version 2.4.

`CakeResponse::charset`(*$charset = null*)
Configure le charset qui sera utilisé dans la réponse.

`CakeResponse::type`(*$contentType = null*)
Configure le type de contenu pour la réponse. Vous pouvez soit utiliser un alias de type de contenu connu, soit le nom du type de contenu complet.

`CakeResponse::cache`(*$since*, *$time = '+1 day'*)
Vous permet de configurer les en-têtes de mise en cache dans la réponse.

`CakeResponse::disableCache`()
Configure les en-têtes pour désactiver la mise en cache des client pour la réponse.

`CakeResponse::sharable`(*$public = null*, *$time = null*)
Configure l'en-tête de Cache-Control pour être soit *public* soit *private* et configure optionnellement une directive de la ressource à un *max-age*. Introduit dans la version 2.1.

`CakeResponse::expires`(*$time = null*)
Permet de configurer l'en-tête *Expires* à une date spécifique. Introduit dans la version 2.1.

`CakeResponse::etag`(*$tag = null*, *$weak = false*)
Configure l'en-tête *Etag* pour identifier de manière unique une ressource de réponse. Introduit dans la version 2.1.

`CakeResponse::modified`(*$time = null*)
Configure l'en-tête *Le-dernier-modifié* à une date et un temps donné dans le format correct. Introduit dans la version 2.1.

`CakeResponse::checkNotModified`(*CakeRequest $request*)
Compare les en-têtes mis en cache pour l'objet request avec l'en-tête mis en cache de la response et détermine si il peut toujours être considéré comme récent. Dans ce cas, il supprime tout contenu de réponse et envoie l'en-tête *304 Not Modified*. Introduit dans la version 2.1.

`CakeResponse::compress`()
Démarre la compression gzip pour la requête.

`CakeResponse::download`(*$filename*)
Vous permet d'envoyer la réponse en pièce jointe et de configurer le nom de fichier.

`CakeResponse::statusCode`(*$code = null*)
Vous permet de configurer le code de statut pour la réponse.

`CakeResponse::body`(*$content = null*)
Configurer le contenu du body pour la réponse.

`CakeResponse::send`()
Une fois que vous avez fini de créer une réponse, appelez send() enverra tous les en-têtes configurés ainsi que le body. Ceci est fait automatiquement à la fin de chaque requête par `Dispatcher`.

`CakeResponse::file`(*$path*, *$options = array()*)
Vous permet de définir un fichier pour l'affichage ou le téléchargement. Introduit dans la version 2.3.

`CakeResponse::param($name)`

Lire les valeurs en toute sécurité dans `$request->params`. Celle-ci enlève la nécessité d'appeler `isset()` ou `empty()` avant l'utilisation des valeurs de param. Introduit dans la version 2.4.

Scaffolding

Une application scaffolding (échafaudage en Français) est une technique permettant au développeur de définir et créer une application qui peut construire, afficher, modifier et détruire facilement des objets. Le Scaffolding dans CakePHP permet également aux développeurs de définir comment les objets sont liés entre eux, et de créer ou casser ces liens.

Pour créer un scaffold, vous n'avez besoin que d'un model et de son controller. Déclarez la variable $scaffold dans le controller, et l'application est déjà prête à tourner !

Le scaffolding par CakePHP est vraiment bien imaginé. Il vous permet de mettre en place une application basique CRUD (Création, Vue, Edition et Destruction) en quelques minutes. Il est si bien fait que vous aurez envie de l'utiliser dans toutes vos applications. Attention ! Nous pensons aussi que le scaffolding est utile, mais veuillez réaliser que ce n'est... qu'un échaufaudage ! C'est une structure très simple à mettre en oeuvre, et il vaut mieux ne l'utiliser qu'au début d'un projet. Il n'a pas été conçu pour être flexible, mais uniquement pour être un moyen temporaire de mettre en place votre application. A partir du moment où vous voudrez adapter les fonctions et les vues associées, il vous faudra désactiver le scaffolding et écrire votre propre code. La console CakePHP bake, que vous pourrez apprendre à connaître dans la prochaine section, est une bonne alternative : il va générer tout le code équivalent à ce que ferait le scaffolding.

Le Scaffolding est à utiliser au tout début du développement d'une application Internet. Le schéma de votre base de données est encore susceptible de changer, ce qui est tout à faire normal à ce stade du processus de création. Ca a un inconvénient : un développeur déteste créer des formulaires dont il ne verra jamais l'utilisation réelle. C'est pour réduire le stress du développeur que le Scaffolding a été introduit dans CakePHP. Il analyse les tables de votre base et crée de façon simple une liste des enregistrements, avec les boutons d'ajout, de suppression et de modification, des formulaires pour l'édition et une vue pour afficher un enregistrement en particulier.

Pour ajouter le Scaffolding dans votre application, ajoutez la variable $scaffold dans votre controller

```
class CategoriesController extends AppController {
    public $scaffold;
}
```

En supposant que vous avez bien crée un model Category dans le bon dossier (/app/Model/Category.php), vous pouvez aller sur http ://exemple.com/categories pour voir votre nouveau scaffold.

Note : Créer des méthodes dans un controller contenant la variable $scaffold peut donner des résultats inattendus. Par exemple, si vous créez une méthode index() dans ce controller, votre méthode remplacera celle rendue normalement par la fonctionnalité de scaffold

---

Le Scaffolding prend bien en compte les relations contenues dans votre model. Ainsi, si votre model Category a une relation BelongsTo avec le model User, vous verrez les identifiants des users dans l'affichage de vos catégories. Puisque scaffolding connaît les associations entre models, vous ne verrez pas d'enregistrements liés dans les vues via scaffold jusqu'à ce que vous ajoutiez manuellement un code d'association

au model. Par exemple, si le model Group hasMany User et que User belongsTo Group, vous devrez ajouter manuellement le code suivant dans vos models User et Group. Avant d'ajouter le code suivant, la vue affiche un select pour le Group dans le nouveau formulaire User :

```
// Dans Group.php
public $hasMany = 'User';
// Dans User.php
public $belongsTo = 'Group';
```

Si vous préférez voir autre chose en plus des identifiants (par exemple les prénoms des users), vous pouvez affecter la variable $displayField dans le model. Voyons comme définir la variable $displayField dans la classe des users, afin que le prénom soit montré en lieu et place de l'unique identifiant. Cette astuce permet de rendre le scaffolding plus lisible dans de nombreux cas :

```
class User extends AppModel {
    public $displayField = 'first_name';
}
```

Créer une interface admin simplifiée avec scaffolding

Si vous avez activé le routage admin dans votre app/config/core.php, avec `Configure::write('Routing.prefixes', array('admin'));` vous pouvez utiliser le scaffolding (échafaudage) pour générer une interface d'administration.

Une fois que vous avez activé le routage admin, assignez votre préfixe d'administration à la variable de scaffolding :

```
public $scaffold = 'admin';
```

Vous serez maintenant capable d'accéder aux actions scaffoldées

```
http://example.com/admin/controller/index
http://example.com/admin/controller/view
http://example.com/admin/controller/edit
http://example.com/admin/controller/add
http://example.com/admin/controller/delete
```

C'est une méthode facile pour créer rapidement une interface d'administration simple. Gardez à l'esprit que vous ne pouvez pas avoir de méthodes de scaffolding à la fois dans la partie admin et dans la partie non-admin en même temps. Comme avec le scaffolding normal, vous pouvez surcharger les méthodes individuelles et les remplacer par vos propres méthodes :

```
public function admin_view($id = null) {
  // du code ici
}
```

Une fois que vous avez remplacé une action de scaffolding, vous devrez créer une vue pour cette action.

Personnaliser les vues obtenues par le Scaffolding

Si vous désirez un rendu un peu différent de vos vues obtenues par le Scaffolding, vous pouvez créer des mises en pages personnalisées. Nous continuons de vous recommander de ne pas utiliser cette technique pour produire vos sites, mais pouvoir modifier les vues peut être utile pour leur développement.

La personnalisation des vues scaffoldées pour un controller spécifique (PostsController dans notre exemple) doit être placée comme ceci :

```
/app/View/Posts/scaffold.index.ctp
/app/View/Posts/scaffold.form.ctp
/app/View/Posts/scaffold.view.ctp
```

Les vues scaffoldées personnalisées pour tous les controllers doivent être placées comme ceci :

```
/app/View/Scaffolds/index.ctp
/app/View/Scaffolds/form.ctp
/app/View/Scaffolds/view.ctp
```

Le Controller Pages

Le cœur de CakePHP est livré avec un controller par défaut `PagesController.php`. C'est un controller simple et optionnel qui permet de rendre un contenu statique. La page d'accueil que vous voyez juste après l'installation est d'ailleurs générée à l'aide de ce controller. Ex : Si vous écrivez un fichier de vue `app/View/Pages/a_propos.ctp`, vous pouvez y accéder en utilisant l'url `http://exemple.com/pages/a_propos`. Vous pouvez modifier le controller Pages selon vos besoins.

Quand vous "cuisinez" une application avec l'utilitaire console de CakePHP, le controller Pages est copié dans votre dossier `app/Controller/` et vous pouvez le modifier selon vos besoins. Ou vous pouvez simplement copier le fichier à partir de `lib/Cake/Console/Templates/skel/Controller/PagesController.php`. Modifié dans la version 2.1 : Avec CakePHP 2.0, le controller Pages était une partie de `lib/Cake`. Depuis 2.1, le controller Pages ne fait plus partie du coeur, mais se situe dans le dossier app.

> **Warning :** Ne modifiez directement AUCUN fichier du dossier `lib/Cake` pour éviter les problèmes lors des mises à jour du coeur dans le futur.

Components (Composants)

Les Components (Composants) sont des regroupements de logique applicative qui sont partagés entre les controllers. Si vous vous surprenez à vouloir copier et coller des choses entre vos controllers, alors vous devriez envisager de regrouper plusieurs fonctionnalités dans un Component.

CakePHP est également livré avec un fantastique ensemble de components, que vous pouvez utiliser pour vous aider :
- Sécurité
- Sessions
- Listes de contrôle d'accès (ACL)

– Emails
– Cookies
– Authentification
– Traitement de requêtes
– Pagination

Chacun de ces components d'origine est détaillé dans des chapitres spécifiques. Pour l'heure, nous allons vous montrer comment créer vos propres components. La création de components vous permet de garder le code de vos controllers propres et vous permet de réutiliser du code entre vos projets.

Configuration des Components

De nombreux components du cœur nécessitent une configuration. Quelques exemples : *Authentification*, *Cookie* et *EmailComponent*. Toute configuration pour ces components, et pour les components en général, se fait dans le tableau des `$components` de la méthode `beforeFilter()` de vos controllers :

```
class PostsController extends AppController {
    public $components = array(
        'Auth' => array(
            'authorize' => array('controller'),
            'loginAction' => array('controller' => 'users', 'action' => 'login')
        ),
        'Cookie' => array('name' => 'CookieMonster')
    );
```

Serait un exemple de configuration d'un component avec le tableau `$components`. Tous les components du coeur permettent aux paramètres d'être configurés dans la méthode de votre controller `beforeFilter()`. C'est utile quand vous avez besoin d'assigner les résultats d'une fonction à la propriété d'un component. Ceci peut aussi être exprimé comme ceci :

```
public function beforeFilter() {
    $this->Auth->authorize = array('controller');
    $this->Auth->loginAction = array('controller' => 'users', 'action' => 'login');

    $this->Cookie->name = 'CookieMonster';
}
```

C'est possible, cependant, que le component nécessite certaines options de configuration avant que le controller `beforeFilter()` soit lancé. Pour cela, certains components permettent aux options de configuration d'être définies dans le tableau `$components` :

```
public $components = array(
    'DebugKit.Toolbar' => array('panels' => array('history', 'session'))
);
```

Consultez la documentation appropriée pour connaître les options de configuration que chaque component fournit.

Un paramètre commun à utiliser est l'option `className`, qui vous autorise les alias des components. Cette fonctionnalité est utile quand vous voulez remplacer `$this->Auth` ou une autre référence de Component commun avec une implémentation sur mesure :

```
// app/Controller/PostsController.php
class PostsController extends AppController {
    public $components = array(
        'Auth' => array(
            'className' => 'MyAuth'
        )
    );
}

// app/Controller/Component/MyAuthComponent.php
App::uses('AuthComponent', 'Controller/Component');
class MyAuthComponent extends AuthComponent {
    // Ajouter votre code pour surcharger le AuthComponent du coeur
}
```

Ce qu'il y a au-dessous donnerait un *alias* `MyAuthComponent` à `$this->Auth` dans vos controllers.

Note : Faire un alias à un component remplace cette instance n'importe où où le component est utilisé, en incluant l'intérieur des autres Components.

Utiliser les Components

Une fois que vous avez inclu quelques components dans votre controller, les utiliser est très simple. Chaque component que vous utilisez est enregistré comme propriété dans votre controller. Si vous avez chargé la `SessionComponent` et le `CookieComponent` dans votre controller, vous pouvez y accéder comme ceci :

```
class PostsController extends AppController {
    public $components = array('Session', 'Cookie');

    public function delete() {
        if ($this->Post->delete($this->request->data('Post.id'))) {
            $this->Session->setFlash('Post deleted.');
            $this->redirect(array('action' => 'index'));
        }
    }
```

Note : Puisque les Models et les Components sont tous deux ajoutés aux controllers en tant que propriété, ils partagent le même 'espace de noms'. Assurez vous de ne pas donner le même nom à un component et à un model.

Charger les components à la volée

Vous n'avez parfois pas besoin de rendre le component accessible sur chaque action. Dans ce cas là, vous pouvez charger à la volée en utilisant la *Component Collection*. A partir de l'intérieur d'un controller, vous pouvez faire comme ce qui suit :

```
$this->OneTimer = $this->Components->load('OneTimer');
$this->OneTimer->getTime();
```

Note : Gardez à l'esprit que le chargement d'un component à la volée ne va pas appeler la méthode initialize. Si le component que vous appelez a cette méthode, vous devrez l'appeler manuellement après le chargement.

Callbacks des Components

Les components vous offrent aussi quelques callbacks durant leur cycle de vie qui vous permettent d'augmenter le cycle de la requête. Allez voir l'api *API de Component* pour plus d'informations sur les callbacks possibles des components.

Créer un Component

Supposons que notre application en ligne ait besoin de réaliser une opération mathématique complexe dans plusieurs sections différentes de l'application. Nous pourrions créer un component pour héberger cette logique partagée afin de l'utiliser dans plusieurs controllers différents.

La première étape consiste à créer un nouveau fichier et une classe pour le component. Créez le fichier dans `/app/Controller/Component/MathComponent.php`. La structure de base pour le component ressemblerait à quelque chose comme ça

```
class MathComponent extends Component {
    public function faireDesOperationsComplexes($montant1, $montant2) {
        return $montant1 + $montant2;
    }
}
```

Note : Tous les components comme Math doivent étendre `Component`. Ne pas le faire vous enverra une exception.

Inclure votre component dans vos controllers

Une fois notre component terminé, nous pouvons l'utiliser au sein des controllers de l'application en plaçant son nom (sans la partie "Component") dans le tableau `$components` du controller. Le controller sera automatiquement pourvu d'un nouvel attribut nommé d'après le component, à travers lequel nous pouvons accéder à une instance de celui-ci :

```
/* Rend le nouveau component disponible par $this->Math
ainsi que le component standard $this->Session */
public $components = array('Math', 'Session');
```

Les Components déclarés dans `AppController` seront fusionnés avec ceux déclarés dans vos autres controllers. Donc il n'y a pas besoin de re-déclarer le même component deux fois.

Quand vous incluez des Components dans un Controller, vous pouvez aussi déclarer un ensemble de paramètres qui seront passés à la méthode initialize() du Component. Ces paramètres peuvent alors être pris en charge par le Component :

```
public $components = array(
    'Math' => array(
        'precision' => 2,
        'generateurAleatoire' => 'srand'
    ),
    'Session', 'Auth'
);
```

L'exemple ci-dessus passerait le tableau contenant "precision" et "generateurAleatoire" comme second paramètre au `MathComponent::__construct()`. Par convention, tout paramètre passé qui est aussi une propriété publique sur votre component aura la valeur basée sur ces paramètres.

Utiliser d'autres Components dans votre Component

Parfois un de vos components a besoin d'utiliser un autre component. Dans ce cas, vous pouvez inclure d'autres components dans votre component exactement de la même manière que dans vos controllers - en utilisant la variable `$components` :

```
// app/Controller/Component/CustomComponent.php
class CustomComponent extends Component {
    // l'autre component que votre component utilise
    public $components = array('Existing');

    public function initialize($controller) {
        $this->Existing->foo();
    }

    public function bar() {
        // ...
   }
}

// app/Controller/Component/ExistingComponent.php
class ExistingComponent extends Component {

    public function initialize($controller) {
        $this->Parent->bar();
    }

    public function foo() {
        // ...
    }
}
```

Notez qu'au contraire d'un component inclu dans un controller, aucun callback ne sera attrapé pour un component inclu dans un component.

API de Component

class `Component`

La classe de base de Component vous offre quelques méthodes pour le chargement facile des autres Components à travers `ComponentCollection` comme nous l'avons traité avec la gestion habituelle des paramètres. Elle fournit aussi des prototypes pour tous les callbacks des components.

`Component::__construct`(*ComponentCollection $collection*, *$settings = array()*)

Les Constructeurs pour la classe de base du component. Tous les `$parametres` qui sont aussi des propriétés publiques, vont avoir leurs valeurs changées pour matcher avec les valeurs de `$settings`.

Les Callbacks

`Component::initialize`(*Controller $controller*)

La méthode initialize est appelée avant la méthode du controller beforeFilter.

`Component::startup`(*Controller $controller*)

La méthode startup est appelée après la méthode du controller beforeFilter mais avant que le controller n'exécute l'action prévue.

`Component::beforeRender`(*Controller $controller*)

La méthode beforeRender est appelée après que le controller exécute la logique de l'action requêtée, mais avant le rendu de la vue et le layout du controller.

`Component::shutdown`(*Controller $controller*)

La méthode shutdown est appelée avant que la sortie soit envoyée au navigateur.

`Component::beforeRedirect`(*Controller $controller*, *$url*, *$status=null*, *$exit=true*)

La méthode beforeRedirect est invoquée quand la méthode de redirection du controller est appelée, mais avant toute action qui suit. Si cette méthode retourne false, le controller ne continuera pas de rediriger la requête. Les variables $url, $status et $exit ont la même signification que pour la méthode du controller. Vous pouvez aussi retourner une chaîne de caractère qui sera interpretée comme une url pour rediriger ou retourner un array associatif avec la clé 'url' et éventuellement 'status' et 'exit'.

CHAPITRE 5

Views (Vues)

Les "Vues" sont le **V** dans MVC. Les vues sont chargées de générer la sortie spécifique requise par la requête. Souvent, cela est fait sous forme HTML, XML ou JSON, mais le streaming de fichiers et la création de PDFs que les utilisateurs peuvent télécharger sont aussi de la responsabilité de la couche View.

CakePHP a quelques classes de vue déjà construites pour gérer les scénarios de rendu les plus communs :
- Pour créer des services web XML ou JSON, vous pouvez utiliser *Vues JSON et XML*.
- Pour servir des fichiers protégés, ou générer des fichiers dynamiquement, vous pouvez utiliser *Vues Media*.
- Pour créer des vues multiples par thème, vous pouvez utliser *Thèmes*.

Templates de Views

La couche view de CakePHP c'est la façon dont vous parlez à vos utilisateurs. La plupart du temps, vos vues afficheront des documents (X)HTML pour les navigateurs, mais vous pourriez aussi avoir besoin de fournir des données AMF à un objet Flash, répondre à une application distante via SOAP ou produire un fichier CSV pour un utilisateur.

Les fichiers de vues de CakePHP sont écrits en pur PHP et ont comme extension par défaut .ctp (Cakephp TemPlate). Ces fichiers contiennent toute la logique de présentation nécessaire à l'organisation des données reçues du contrôleur, dans un format qui satisfasse l'audience que vous recherchez. Si vous préfèrez utiliser un langage de template comme Twig, ou Smarty, une sous-classe de View fera le pont entre votre language de template et CakePHP.

Les fichiers de vues sont stockées dans `/app/View/`, dans un dossier portant le nom du contrôleur qui utilise ces fichiers et le nom de la vue correspondante. Par exemple, l'action "view()" du contrôleur Produits devrait normalement se trouver dans `/app/View/Products/view.ctp`.

La couche vue de CakePHP peut être constituée d'un certain nombre de parties différentes. Chaque partie a différent usages qui seront présentés dans ce chapitre :
- **views** : Les Views sont la partie de la page qui est unique pour l'action lancée. Elles sont la substance de la réponse de votre application.

- **elements** : morceaux de code de view plus petits, réutilisables. Les éléments sont habituellement rendus dans les vues.
- **layouts** : fichiers de vue contenant le code de présentation qui se retrouve dans plusieurs interfaces de votre application. La plupart des vues sont rendues à l'intérieur d'un layout.
- **helpers** : ces classes encapsulent la logique de vue qui est requise à de nombreux endroits de la couche vue. Parmi d'autres choses, les helpers (assistants) de CakePHP peuvent vous aider à créer des formulaires, des fonctionnalités AJAX, de paginer les données du modèle ou à délivrer des flux RSS.

Views étendues

Introduit dans la version 2.1. Une vue étendue vous permet d'enrouler une vue dans une autre. En combinant cela avec *view blocks*, cela vous donne une façon puissante pour garder vos vues *DRY*. Par exemple, votre application a une sidebar qui a besoin de changer selon la vue spécifique en train d'être rendue. En étendant un fichier de vue commun, vous pouvez éviter de répeter la balise commune pour votre sidebar, et seulement définir les parties qui changent :

```
// app/View/Common/view.ctp
<h1><?php echo $this->fetch('title'); ?></h1>
<?php echo $this->fetch('content'); ?>

<div class="actions">
    <h3>Related actions</h3>
    <ul>
    <?php echo $this->fetch('sidebar'); ?>
    </ul>
</div>
```

Le fichier de vue ci-dessus peut être utilisé comme une vue parente. Il s'attend à ce que la vue étendue, il va définir des blocks `sidebar` et `title`. Le block `content` est un block spécial que CakePHP crée. Il contiendra tous les contenus non capturés de la vue étendue. En admettant que notre fichier de vue a une variable `$post` avec les données sur notre post. Notre vue pourrait ressembler à ceci :

```
<?php
// app/View/Posts/view.ctp
$this->extend('/Common/view');

$this->assign('title', $post);

$this->start('sidebar');
?>
<li>
<?php
echo $this->Html->link('edit', array(
    'action' => 'edit',
    $post['Post']['id']
)); ?>
</li>
<?php $this->end(); ?>

// Le contenu restant sera disponible en tant que block 'content'
```

```
// dans la vue parente.
echo h($post['Post']['body']);
```

L'exemple ci-dessus vous montre comment vous pouvez étendre une vue, et remplir un ensemble de bloc. Tout contenu qui ne serait pas déjà dans un bloc défini, sera capturé et placé dans un bloc spécial appellé `content`. Quand une vue contient un appel vers un `extend()`, l'éxécution continue jusqu'à la fin de la vue actuelle. Une fois terminé, la vue étendue va être générée. En appellant `extend()` plus d'une fois dans un fichier de vue, le dernier appel va outrepasser les précédents :

```
$this->extend('/Common/view');
$this->extend('/Common/index');
```

Le code précédent va définir `/Common/index.ctp` comme étant la vue parente de la vue actuelle.

Vous pouvez imbriquer les vues autant que vous le voulez et que cela vous est nécessaire. Chaque vue peut étendre une autre vue si vous le souhaitez. Chaque vue parente va récupérer le contenu de la vue précédente en tant que bloc `content`.

Note : Vous devriez éviter d'utiliser `content` comme nom de bloc dans votre application. CakePHP l'utilise pour définir le contenu non-capturé pour les vues étendues.

Utiliser les blocs de vues

Introduit dans la version 2.1. Les blocs de vue remplacent les `$scripts_for_layout` et fournissent une API flexible qui vous permet de définir des slots (emplacements), ou blocs, dans vos vues / layouts qui peuvent être définies ailleurs. Par exemple, les blocs pour implémenter des choses telles que les sidebars, ou des régions pour charger des ressources dans l'en-tête / pied de page du layout. Un bloc peut être définit de deux manières. Soit en tant que bloc capturant, soit en le déclarant explicitement. Les méthodes `start()`, `append()` et `end()` vous permettent de travailler avec les blocs capturant.

> // Creer le bloc sidebar. $this->start('sidebar') ; echo $this->element('sidebar/recent_topics') ; echo $this->element('sidebar/recent_comments') ; $this->end() ;
>
> // Le rattacher a la sidebar plus tard. $this->append('sidebar') ; echo $this->element('sidebar/popular_topics') ; $this->end() ;

Vous pouvez aussi le rattacher à l'intérieur d'un bloc en utilisant `start()` pluieurs fois. La méthode `assign()` peut être utilisée pour nettoyer ou outrepasser un bloc à n'importe quel moment :

```
// Nettoyer le contenu précedent de la sidebar.
$this->assign('sidebar', '');
```

Dans 2.3, certaines nouvelles méthodes ont été ajoutées pour travailler avec les blocs. Le `prepend()` pour ajouter du contenu avant un bloc existant :

```
// Ajoutez avant la sidebar
$this->prepend('sidebar', 'ce contenu va au-dessus de la sidebar');
```

La méthode `startIfEmpty()` peut être utilisée pour commencer un bloc **seulement** si il est vide ou non défini. Si le bloc existe déjà, le contenu capturé va être écarté. C'est utile quand vous voulez définir le contenu par défaut de façon conditionnel pour un bloc, qui ne doit pas déjà exister :

```
.. code-block:: php
```

> // Dans un fichier de vue. // Crée un bloc de navbar $this->startIfEmpty('navbar') ; echo $this->element('navbar') ; echo $this->element('notifications') ; $this->end() ;

```
// Dans une vue/layout parente
<?php $this->startIfEmpty('navbar'); ?>
<p>Si le block n est pas défini pour l instant - montrer ceci à la place</p>
<?php $this->end(); ?>

// Quelque part plus loin dans la vue/layout parent
echo $this->fetch('navbar');
```

Dans l'exemple ci-dessus, le bloc `navbar` va seulement contenir le contenu ajouté dans la première section. Puisque le bloc a été défini dans la vue enfant, le contenu par défaut avec la balise `<p>` sera écarté.

Note : Vous devriez éviter d'utiliser `content` comme nom de bloc. Celui-ci est utilisé par CakePHP en interne pour étendre les vues, et le contenu des vues dans le layout.

Afficher les blocs

Introduit dans la version 2.1. Vous pouvez afficher les blocs en utilisant la méthode `fetch()`. Cette dernière va, de manière sécurisée, générer un bloc, en retournant '' si le bloc n'existe pas :

```
echo $this->fetch('sidebar');
```

Vous pouvez également utiliser fetch pour afficher du contenu, sous conditions, qui va entourer un bloc existant. Ceci est très utile dans les layouts, ou dans les vues étendues où vous voulez, sous conditions, afficher des en-têtes ou autres balises :

```
// dans app/View/Layouts/default.ctp
<?php if ($this->fetch('menu')): ?>
<div class="menu">
    <h3>Menu options</h3>
    <?php echo $this->fetch('menu'); ?>
</div>
<?php endif; ?>
```

Depuis 2.3.0, vous pouvez aussi fournir une valeur par défaut pour un bloc qui ne devrait pas avoir de contenu. Cela vous permet d'ajouter facilement du contenu placeholder, pour des déclarations vides. Vous pouvez fournir une valeur par défaut en utilisant le 2ème argument :

```
<div class="shopping-cart">
    <h3>Your Cart</h3>
    <?php echo $this->fetch('cart', 'Votre cart est vide'); ?>
</div>
```

Modifié dans la version 2.3 : L'argument `$default` a été ajouté dans 2.3.

Utiliser des blocks pour les fichiers de script et les css

Introduit dans la version 2.1. Les Blocks remplacent la variable de layout `$scripts_for_layout` qui est dépréciée. A la place, vous devrez utiliser les blocks. `HtmlHelper` lie dans les blocks de vues avec les méthodes `script()`, `css()`, et `meta()` qui chacune met à jour un block avec le même nom quand l'option `inline = false` est utilisée :

```
<?php
// dans votre fichier de vue
$this->Html->script('carousel', array('inline' => false));
$this->Html->css('carousel', null, array('inline' => false));
?>

// dans votre fichier de layout.
<!DOCTYPE html>
<html lang="en">
    <head>
    <title><?php echo $this->fetch('title'); ?></title>
    <?php echo $this->fetch('script'); ?>
    <?php echo $this->fetch('css'); ?>
    </head>
    // rest du layout suit
```

Le `HtmlHelper` vous permet aussi de contrôler vers quels blocks vont les scripts :

```
// dans votre vue
$this->Html->script('carousel', array('block' => 'scriptBottom'));

// dans votre layout
echo $this->fetch('scriptBottom');
```

Layouts

Un layout contient le code de présentation qui entoure une vue. Tout ce que vous voulez voir dans toutes vos vues devra être placé dans un layout.

Le fichier de layout par défaut de CakePHP est placé dans `/app/View/Layouts`. Si vous voulez changer entièrement le look de votre application, alors c'est le bon endroit pour commencer, parce que le code de vue de rendu du controller est placé à l'intérieur du layout par défaut quand la page est rendue.

Les autres fichiers de layout devront être placés dans `/app/View/Layouts`. Quand vous créez un layout, vous devez dire à CakePHP où placer la sortie pour vos vues. Pour ce faire, assurez-vous que votre layout contienne `$this->fetch('content')`. Voici un exemple auquel un layout pourrait ressembler :

```
<!DOCTYPE html>
<html lang="en">
<head>
<title><?php echo $title_for_layout?></title>
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
<!-- Include external files and scripts here (See HTML helper for more info.) -->
echo $this->fetch('meta');
echo $this->fetch('css');
echo $this->fetch('script');
?>
</head>
<body>

<!-- Si vous voulez afficher une sorte de menu pour toutes vos vues, mettez
le ici -->
<div id="header">
    <div id="menu">...</div>
</div>

<!-- Voilà l'endroit ou je souhaite que mes vues soient affichées -->
<?php echo $this->fetch('content'); ?>

<!-- Ajoute un footer sur chaque page affichée -->
<div id="footer">...</div>

</body>
</html>
```

Note : Avant la version 2.1, la méthode fetch() n'était pas disponible, `fetch('content')` remplace `$content_for_layout` et les lignes `fetch('meta')`, `fetch('css')` et `fetch('script')` étaient contenues dans la variable `$scripts_for_layout` dans la version 2.0.

Les blocks `script`, `css` et `meta` contiennent tout contenu défini dans les vues en utilisant le helper HTML intégré. Il est utile pour inclure les fichiers javascript et les CSS à partir des vues.

Note : Quand vous utilisez `HtmlHelper::css()` ou `HtmlHelper::script()` dans les fichiers de vues, spécifiez 'false' dans l'option 'inline' option pour placer la source html dans un block avec le même nom. (Regardez l'API pour plus de détails sur leur utilisation).

Le block `content` contient les contenus de la vue rendue.

`$title_for_layout` contient le titre de la page. Cette variable est générée automatiquement, mais vous pouvez la surcharger en la configurant dans votre controller/view.

Pour définir le titre pour le layout, il est plus facile de le faire dans le controller, en configurant la variable `$title_for_layout` :

```
class UsersController extends AppController {
    public function view_active() {
        $this->set('title_for_layout', 'Voir les Utilisateurs actifs');
```

```
    }
}
```

Vous pouvez aussi définir la variable title_for_layout depuis l'intérieur d'un fichier de vue :

```
$this->set('title_for_layout', $titleContent);
```

Vous pouvez créer autant de layouts que vous souhaitez : placez les juste dans le répertoire `app/View/Layouts`, et passez de l'un à l'autre depuis les actions de votre controller en utilisant la propriété `$layout` de votre controller ou de votre vue :

```
// A partir d'un controller
public function admin_view() {
    // stuff
    $this->layout = 'admin';
}

// A partir d'un fichier de vue
$this->layout = 'loggedin';
```

Par exemple, si une section de mon site incorpore un plus petit espace pour une bannière publicitaire, je peux créer un nouveau layout avec le plus petit espace de publicité et le spécifier comme un layout pour toutes les actions du controller en utilisant quelque chose comme :

```
class UsersController extends AppController {
    public function view_active() {
        $this->set('title_for_layout', 'Voir les Utilisateurs actifs');
        $this->layout = 'default_small_ad';
    }

    public function view_image() {
        $this->layout = 'image';
        //sort une image de l\'utilisateur
    }
}
```

CakePHP dispose de deux fonctionnalités de layout dans le coeur (en plus du layout default de CakePHP) vous pouvez utiliser dans votre propre application : 'ajax' et 'flash'. Le layout Ajax est pratique pour élaborer des réponses Ajax - c'est un layout vide (la plupart des appels ajax ne nécessitent qu'un peu de balise en retour, et pas une interface de rendu complète). Le layout flash est utilisé pour les messages montrés par la méthode `Controller::flash()`.

Trois autres layouts, xml, js, et rss, existent dans le coeur pour une façon rapide et facile pour servir du contenu qui n'est pas du text/hmtl.

Utiliser les layouts à partir de plugins

Introduit dans la version 2.1. Si vous souhaitez utiliser un layout qui existe dans un plugin, vous pouvez utiliser *syntaxe de plugin*. Par exemple pour utiliser le layout de contact à partir du plugin Contacts :

```
class UsersController extends AppController {
    public function view_active() {
        $this->layout = 'Contacts.contact';
    }
}
```

Elements

Beaucoup d'applications ont des petits blocks de code de présentation qui doivent être répliqués d'une page à une autre, parfois à des endroits différents des le layout. CakePHP peut vous aider à répéter des parties de votre site web qui doivent être réutilisées. Ces parties réutilisables sont appelées des Elements. Les Publicités, les boites d'aides, les contrôles de navigation, les menus supplémentaires, les formulaires de connexion et de sortie sont souvent intégrés dans CakePHP en elements. Un element est tout bêtement une mini-vue qui peut être inclue dans d'autres vues, dans les layouts, et même dans d'autres elements. Les elements peuvent être utilisés pour rendre une vue plus lisible, en plaçant le rendu d'éléments répétitifs dans ses propres fichiers. Ils peuvent aussi vous aider à réutiliser des fragments de contenu dans votre application.

Les elements se trouvent dans le dossier `/app/View/Elements/`, et ont une extension .ctp. Ils sont affichés en utilisant la méthode element de la vue :

```
echo $this->element('helpbox');
```

Passer des Variables à l'intérieur d'un Element

Vous pouvez passer des données dans un element grâce au deuxième argument de element :

```
echo $this->element('helpbox', array(
    "helptext" => "Oh, this text is very helpful."
));
```

Dans le fichier element, toutes les variables passés sont disponibles comme des membres du paramètre du tableau (de la même manière que `Controller::set()` fonctionne dans le controller avec les fichiers de vues). Dans l'exemple ci-dessus, le fichier `/app/View/Elements/helpbox.ctp` peut utiliser la variable `$helptext` :

```
// A l'intérieur de app/View/Elements/helpbox.ctp
echo $helptext; //outputs "Oh, this text is very helpful."
```

La méthode `View::element()` supporte aussi les options pour l'element. Les options supoortés sont 'cache' et 'callbacks'. Un exemple :

```
echo $this->element('helpbox', array(
        "helptext" => "Ceci est passé à l'element comme $helptext",
        "foobar" => "Ceci est passé à l'element via $foobar",
    ),
    array(
        "cache" => "long_view", // utilise la configuration de cache "long_view"
        "callbacks" => true // défini à true pour avoir before/afterRender appelé pour l'e
```

```
    )
);
```

La mise en cache d'element est facilitée par la classe `Cache`. Vous pouvez configurer les elements pour être stockés dans toute configuration de Cache que vous avez défini. Cecla vous donne une grande flexibilité pour choisir où et combien de temps les elements sont stockés. Pour mettre en cache les différentes versions du même element dans une application, fournissez une valeur de clé cache unique en utilisant le format suivant :

```
$this->element('helpbox', array(), array(
        "cache" => array('config' => 'short', 'key' => 'unique value')
    )
);
```

Vous pouvez tirer profit des elements en utilisant `requestAction()`. La fonction `requestAction()` récupère les variables de vues à partir d'une action d'un controller et les retourne en tableau. Cela permet à vos elements de fonctionner dans un style MVC pur. Créez une action du controller qui prépare les variables de la vue pour vos elements, ensuite appelez `requestAction()` depuis l'intérieur du deuxième paramètre de `element()` pour alimenter en variables de vues l'element depuis votre controller.

Pour ce faire, ajoutez quelque chose comme ce qui suit dans votre controller, en reprenant l'exemple du Post :

```
class PostsController extends AppController {
    // ...
    public function index() {
        $posts = $this->paginate();
        if ($this->request->is('requested')) {
            return $posts;
        } else {
            $this->set('posts', $posts);
        }
    }
}
```

Et ensuite dans l'element, nous pouvons accéder au model des posts paginés. Pour obtenir les cinq derniers posts dans une liste ordonnée, nous ferions ce qui suit :

```
<h2>Derniers Posts</h2>
<?php $posts = $this->requestAction('posts/index/sort:created/direction:asc/limit:5'); ?>
<ol>
<?php foreach ($posts as $post): ?>
    <li><?php echo $post['Post']['title']; ?></li>
<?php endforeach; ?>
</ol>
```

Mise en cache des Elements

Vous pouvez tirer profit de la mise en cache de vue de CakePHP si vous fournissez un paramètre cache. Si défini à true, cela va mettre en cache l'element dans la configuration 'default' de Cache. Sinon, vous pouvez

définir quelle configuration de cache doit être utilisée. Regardez *La mise en cache* pour plus d'informations sur la façon de configurer `Cache`. Un exemple simple de mise en cache d'un element serait par exemple :

```
echo $this->element('helpbox', array(), array('cache' => true));
```

Si vous rendez le même element plus d'une fois dans une vue et que vous avez activer la mise en cache, assurez-vous de définir le paramètre 'key' avec un nom différent à chaque fois. Cela évitera que chaque appel successif n'écrase le résultat de la mise en cache du précédent appel de element(). Par exemple :

```
echo $this->element(
    'helpbox',
    array('var' => $var),
    array('cache' => array('key' => 'first_use', 'config' => 'view_long')
);

echo $this->element(
    'helpbox',
    array('var' => $differenVar),
    array('cache' => array('key' => 'second_use', 'config' => 'view_long')
);
```

Ce qui est au-dessus va s'enquérir que les deux résultats d'element sont mis en cache séparément. Si vous voulez que tous les elements mis en cache utilisent la même configuration du cache, vous pouvez sauvegarder quelques répétitions, en configurant `View::$elementCache` à la configuration de Cache que vous souhaitez utiliser. CakePHP va utiliser cette configuration, quand aucune n'est donnée.

Requêter les Elements à partir d'un Plugin

2.0

Pour charger un element d'un plugin, utilisez l'option *plugin* (enlevé de l'option *data* dans 1.x) :

```
echo $this->element('helpbox', array(), array('plugin' => 'Contacts'));
```

2.1

Si vous utilisez un plugin et souhaitez utiliser les elements à partir de l'intérieur d'un plugin, utilisez juste la *syntaxe de plugin* habituelle. Si la vue est rendue pour un controller/action d'un plugin, le nom du plugin va automatiquement être préfixé pour tous les elements utilisés, à moins qu'un autre nom de plugin ne soit présent. Si l'element n'existe pas dans le plugin, il ira voir dans le dossier principal APP. :

```
echo $this->element('Contacts.helpbox');
```

Si votre vue fait parti d'un plugin, vous pouvez ne pas mettre le nom du plugin. Par exemple, si vous êtes dans le `ContactsController` du plugin Contacts :

```
echo $this->element('helpbox');
// et
echo $this->element('Contacts.helpbox');
```

Sont équivalents et résulteront au même element rendu. Modifié dans la version 2.1 : L'option `$options[plugin]` a été déprécié et le support pour `Plugin.element` a été ajouté.

Créer vos propres classes de vue

Vous avez peut-être besoin de créer vos propres classes de vue pour activer des nouveaux types de données de vue, ou ajouter de la logique supplémentaire de rendu de vue personnalisée. Comme la plupart des components de CakePHP, les classes de vue ont quelques conventions :

- Les fichiers de classe de View doivent être mis dans `App/View`. Par exemple `App/View/PdfView.php`.
- Les classes de View doivent être suffixées avec `View`. Par exemple `PdfView`.
- Quand vous référencez les noms de classe de vue, vous devez omettre le suffixe `View`. Par exemple `$this->viewClass = 'Pdf';`.

Vous voudrez aussi étendre `View` pour vous assurer que les choses fonctionnent correctement :

```
// dans App/View/PdfView.php

App::uses('View', 'View');
class PdfView extends View {
    public function render($view = null, $layout = null) {
        // logique personnalisée ici.
    }
}
```

Remplacer la méthode render vous laisse le contrôle total sur la façon dont votre contenu est rendu.

API de View

class `View`

Les méthodes de View sont accessibles dans toutes les vues, element et fichiers de layout. Pour appeler toute méthode de view, utilisez `$this->method()`

`View::set` (*string $var*, *mixed $value*)

Les Views ont une méthode `set()` qui est analogue à `set()` qui se trouvent dans les objets du controller. Utiliser set() à partir de votre fichier de vue va ajouter les variables au layout et elements qui seront rendus plus tard. Regarder *Les Méthodes du Controller* pour plus d'informations sur l'utilisation de set().

Dans votre fichier de vue, vous pouvez faire :

```
$this->set('activeMenuButton', 'posts');
```

Ensuite dans votre fichier de layout la variable `$activeMenuButton` sera disponible et contiendra la valeur 'posts'.

`View::getVar` (*string $var*)

Récupère la valeur de viewVar avec le nom $var

View::**getVars**()
Récupère une liste de toutes les variables de view disponibles dans le cadre de rendu courant. Retourne un tableau des noms de variable.

View::**element**(*string $elementPath*, *array $data*, *array $options = array()*)
Rend un element ou une vue partiel. Regardez la section sur *Elements* pour plus d'informations et d'exemples.

View::**uuid**(*string $object*, *mixed $url*)
Génére un ID de DOM unique pas au hasard pour un objet, basé sur le type d'objet et l'url. Cette méthode est souvent utilisée par les helpers qui ont besoin de générer un ID de DOM unique pour les elements comme le `JsHelper` :

```
$uuid = $this->uuid('form', array('controller' => 'posts', 'action' => 'index'));
//$uuid contient 'form0425fe3bad'
```

View::**addScript**(*string $name*, *string $content*)
Ajoute du contenu au buffer des scripts internes. Ce buffer est rendu disponible dans le layout dans `$scripts_for_layout`. Cette méthode est utile quand vous créez des helpers qui ont besoin d'ajouter du javascript ou du css directement au layout. Gardez à l'esprit que les scripts ajoutés à partir du layout, ou des elements du layout ne seront pas ajoutés à `$scripts_for_layout`. Cette méthode est plus souvent utilisée de l'intérieur des helpers, comme pour les helpers *JSHelper* et *HTMLHelper*. Obsolète depuis la version 2.1 : Utilisez les fonctionnalités *Utiliser les blocs de vues* à la place.

View::**blocks**()
Récupère les noms de tous les blocks définis en tant que tableau.

View::**start**(*$name*)
Commence un block de capture pour un block de vue. Regardez la section sur les *Utiliser les blocs de vues* pour avoir des exemples. Introduit dans la version 2.1.

View::**end**()
Termine le block de capture ouvert le plus en haut. Regardez la section sur les *Utiliser les blocs de vues* pour avoir des exemples. Introduit dans la version 2.1.

View::**append**(*$name*, *$content*)
Ajoute dans le block avec `$name`. Regardez la section sur les *Utiliser les blocs de vues* pour des exemples. Introduit dans la version 2.1.

View::**prepend**(*$name*, *$content*)
Ajoute avant dans le block avec `$name`. Regardez la section *Utiliser les blocs de vues* pour des exemples. Introduit dans la version 2.3.

View::**startIfEmpty**(*$name*)
Commence un block sous conditions, seulement si il est vide. Tout le contenu dans le block va être capturé et écarté si le block est déjà défini. Introduit dans la version 2.3.

View::**assign**(*$name*, *$content*)
Assigne la valeur d'un block. Cela va surcharger tout contenu existant. Regardez la section sur les *Utiliser les blocs de vues* pour des exemples. Introduit dans la version 2.1.

View::**fetch**(*$name*, *$default* = '')
Récupère la valeur d'un block. Si un block est vide ou non défini, '' va être retourné. Regardez la section sur les *Utiliser les blocs de vues* pour des exemples. Introduit dans la version 2.1.

View::**extend**(*$name*)
Etend la vue/element/layout courant avec celle dans $name. Regardez la section sur les *Views étendues* pour les examples. Introduit dans la version 2.1.

property View::$**layout**
Définissez le layout qui va entouré la vue courante.

property View::$**elementCache**
La configuration de cache utilisée pour les elements de cache. Définir cette propriété va changer la configuration par défaut utilisée pour mettre en cache les elements. Celle par défaut peut être surchargée en utilisant l'option 'cache' dans la méthode element.

property View::$**request**
Une instance de `CakeRequest`. Utilisez cette instance pour accéder aux informations qui concernent la requête courante.

property View::$**output**
Contient le dernier contenu rendu d'une view, ou d'un fichier de view, ou d'un contenu de layout. Obsolète depuis la version 2.1 : Utilisez `$view->Blocks->get('content');` à la place.

property View::$**Blocks**
Une instance de `ViewBlock`. Utilisé pour fournir la fonctionnalité des blocks de view dans le rendu de view. Introduit dans la version 2.1.

En savoir plus sur les vues

Thèmes

Vous pouvez profiter des thèmes, ce qui facilite le changement du visuel et du ressenti de votre page rapidement et facilement.

Pour utiliser les thèmes, spécifiez le nom du thème dans votre controller :

```
class ExempleController extends AppController {
    public $theme = 'Exemple';
}

.. versionchanged:: 2.1
    Les versions antérieures à 2.1 ont besoin de définir
    ``$this->viewClass = 'Theme'``.
    2.1 enlève cette condition puisque la classe normale ``View``
    supporte les thèmes.
```

Vous pouvez également définir ou modifier le nom du thème dans une action ou dans les fonctions de callback `beforeFilter` ou `beforeRender` :

```
$this->theme = 'AutreExemple';
```

Les fichiers de vue du thème ont besoin d'être dans le dossier `/app/View/Themed/`. Dans le dossier du thème, créez un dossier utilisant le même nom que le nom de votre thème. Par exemple, Le thème ci-dessus serait trouvé dans `/app/View/Themed/AutreExemple`. Il est important de se souvenir que CakePHP attend des noms de thème en CamelCase. Au-delà de ça, la structure de dossier dans le dossier `/app/View/Themed/Exemple/` est exactement le même que `/app/View/`.

Par exemple, le fichier de vue pour une action edit d'un contôleur Posts residera dans `/app/View/Themed/Exemple/Posts/edit.ctp`. Les fichiers de Layout résideront dans `/app/View/Themed/Exemple/Layouts/`.

Si un fichier de vue ne peut pas être trouvé dans le thème, CakePHP va essayer de localiser le fichier de vue dans le dossier `/app/View/`. De cette façon, vous pouvez créer des fichiers de vue master et simplement les remplacer au cas par cas au sein de votre dossier de thème.

Assets du thème

Les thèmes peuvent contenir des assets statiques ainsi que des fichiers de vue. Un thème peut inclure tout asset nécessaire dans son répertoire webroot. Cela permet un packaging facile et une distribution des thèmes. Pendant le développement, les requêtes pour les assets du thème seront gérés par `Dispatcher`. Pour améliorer la performance des environnements de production, il est recommandé, soit que vous fassiez un lien symbolique, soit que vous copiez les assets du thème dans le webroot de application. Voir ci-dessous pour plus d'informations.

Pour utiliser le nouveau thème, créez des répertoires de type `app/View/Themed/<nomDuTheme>/webroot<chemin_vers_fichier>` dans votre thème. Le Dispatcher se chargera de trouver les assets du thème corrects dans vos chemins de vue.

Tous les helpers integrés dans CakePHP ont intégrés l'existence des thèmes et vont créer des chemins d'accès corrects automatiquement. Comme pour les fichiers de vue, si un fichier n'est pas dans le dossier du thème, il sera par défaut dans le dossier principal webroot

```
//Quand dans un thème avec un nom de 'purple_cupcake'
$this->Html->css('main.css');

//crée un chemin comme
/theme/purple_cupcake/css/main.css

//et fait un lien vers
app/View/Themed/PurpleCupcake/webroot/css/main.css
```

Augmenter la performance des assets du plug-in et du thème

C'est un fait bien connu que de servir les assets par le biais de PHP est assuré d'être plus lent que de servir ces assets sans invoquer PHP. Et tandis que l'équipe du coeur a pris des mesures pour rendre le plugin et l'asset du thème servis aussi vite que possible, il peut y avoir des situations où plus de performance est requis. Dans ces situations, il est recommandé soit que vous fassiez un lien symbolique soit que vous

fassiez une copie sur les assets du plug-in/thème à des répertoires dans `app/webroot` avec des chemins correspondant à ceux utilisés par cakephp.
– `app/Plugin/DebugKit/webroot/js/my_file.js` devient `app/webroot/DebugKit/js/my_file.js`.
– `app/View/Themed/Navy/webroot/css/navy.css` devient `app/webroot/theme/Navy/css/navy.cs`

Vues Media

class `MediaView`

Obsolète depuis la version 2.3 : Utilisez *Envoyer des pièces jointes* à la place. Les vues Media vous permettent d'envoyer des fichiers binaires à l'utilisateur. Par exemple, vous souhaiteriez avoir un répertoire de fichiers en dehors de webroot pour empêcher les utilisateurs de faire un lien direct sur eux. Vous pouvez utiliser la vue Media pour tirer le fichier à partir d'un fichier spécial dans /app/, vous permettant d'améliorer l'authentification avant la livraison du fichier à l'utilisateur.

Pour utiliser la vue Media, vous avez besoin de dire à votre controller d'utiliser la classe MediaView au lieu de la classe View par défaut. Après ça, passez juste les paramètres en plus pour spécifier où votre fichier se trouve :

```
class ExempleController extends AppController {
    public function download() {
        $this->viewClass = 'Media';
        // Download app/outside_webroot_dir/example.zip
        $params = array(
            'id'        => 'example.zip',
            'name'      => 'example',
            'download'  => true,
            'extension' => 'zip',
            'path'      => APP . 'outside_webroot_dir' . DS
        );
        $this->set($params);
    }
}
```

Ici vous trouvez un exemple de rendu d'un fichier qui a un type mime qui n'est pas inclu dans le tableau `$mimeType` de MediaView. Nous utilisons aussi un chemin relatif qui va être par défaut dans votre dossier `app/webroot` :

```
public function download() {
    $this->viewClass = 'Media';
    // Render app/webroot/files/example.docx
    $params = array(
        'id'        => 'example.docx',
        'name'      => 'example',
        'extension' => 'docx',
        'mimeType'  => array(
            'docx' => 'application/vnd.openxmlformats-officedocument.wordprocessingml.docu
        ),
        'path'      => 'files' . DS
    );
    $this->set($params);
}
```

Paramètres configurables

id L'ID est le nom du fichier tel qu'il réside sur le serveur de fichiers, y compris l'extension de fichier.

name Le nom vous permet de spécifier un nom de fichier alternatif à envoyer à l'utilisateur. Spécifiez le nom sans l'extension du fichier.

download Une valeur boléenne indiquant si les en-têtes doivent être définis pour forcer le téléchargement.

extension L'extension du fichier. Ceci est en correspondance avec une liste interne de types mime acceptables. Si le type MIME spécifié n'est pas dans la liste (ou envoyé dans le tableau de paramètres mimeType), le fichier ne sera pas téléchargé.

path Le nom du dossier, y compris le séparateur de répertoire finale. Le chemin doit être absolu, mais peut être par rapport au dossier `app/webroot`.

mimeType Un tableau avec des types MIME supplémentaires à fusionner avec une liste interne dans MediaView de types mime acceptables.

cache Une valeur booléenne ou entière - Si la valeur est vraie, elle permettra aux navigateurs de mettre en cache le fichier (par défaut à false si non définie), sinon réglez le sur le nombre de secondes dans le futur pour lorsque le cache expirera.

Vues JSON et XML

Deux nouvelles classes de vue dans CakePHP 2.1. Les vues `XmlView` et `JsonView` vous laissent créer facilement des réponses XML et JSON, et sont intégrées avec `RequestHandlerComponent`.

En activant `RequestHandlerComponent` dans votre application, et en activant le support pour les extensions `xml` et/ou `json`, vous pouvez automatiquement vous appuyer sur les nouvelles classes de vue. `XmlView` et `JsonView` feront référence aux vues de données pour le reste de cette page.

Il y a deux façons de générer des vues de données. La première est en utilisant la clé `_serialize`, et la seconde en créant des fichiers de vue normaux.

Activation des vues de données dans votre application

Avant que vous puissiez utiliser les classes de vue de données, vous aurez besoin de faire un peu de configuration :

1. Activez les extensions json et/ou xml avec `Router::parseExtensions()`. Cela activera Router pour gérer les multiples extensions.
2. Ajoutez le `RequestHandlerComponent` à la liste de components de votre controller. Cela activera automatiquement le changement de la classe de vue pour les types de contenu.

Introduit dans la version 2.3 : La méthode `RequestHandlerComponent::viewClassMap()` a été ajoutée pour lier les types aux viewClasses. La configuration de viewClassMap ne va pas fonctionner avec les versions précédentes. Après avoir ajouté `Router::parseExtensions('json');` à votre fichier de routes, CakePHP changera automatiquement les classes de vue quand une requête sera faite avec l'extension `.json`, ou quand l'en-tête Accept sera `application/json`.

Utilisation des vues de données avec la clé _serialize

La clé `_serialize` est une variable de vue spéciale qui indique quel autre(s) variable(s) de vue devraient être sérialisée(s) quan on utilise la vue de données. Cela vous permet de sauter la définition des fichiers de vue pour vos actions de controller si vous n'avez pas besoin de faire un formatage avant que vos données soient converties en json/xml.

Si vous avez besoin de faire n'importe quel formatage ou manipulation de vos variables de vue avant la génération de la réponse, vous devriez utiliser les fichiers de vue. La valeur de `_serialize` peut être soit une chaîne de caractère, soit un tableau de variables de vue pour sérialiser :

```
class PostsController extends AppController {
    public function index() {
        $this->set('posts', $this->paginate());
        $this->set('_serialize', array('posts'));
    }
}
```

Vous pouvez aussi définir `_serialize` en tableau de variables de vue pour combiner :

```
class PostsController extends AppController {
    public function index() {
        // some code that created $posts and $comments
        $this->set(compact('posts', 'comments'));
        $this->set('_serialize', array('posts', 'comments'));
    }
}
```

Définir `_serialize` en tableau a le bénéfice ajouté d'ajouter automatiquement un elément de top-niveau `<response>` en utilisant `XmlView`. Si vous utilisez une valeur de chaîne de caractère pour `_serialize` et XmlView, assurez vous que vos variables de vue aient un elément unique de top-niveau. Sans un elément de top-niveau, le Xml ne pourra être généré.

Utilisation d'une vue de données avec les fichiers de vue

Vous devriez utiliser les fichiers de vue si vous avez besoin de faire des manipulations du contenu de votre vue avant de créer la sortie finale. Par exemple, si vous avez des posts, qui ont un champ contenant du HTML généré, nous voudrons probablement omettre ceci à partir d'une réponse JSON. C'est une situation où un fichier de vue serait utile :

```
// Code du controller
class PostsController extends AppController {
    public function index() {
        $this->set(compact('posts', 'comments'));
    }
}

// Code de la vue - app/View/Posts/json/index.ctp
foreach ($posts as &$post) {
    unset($post['Post']['generated_html']);
```

```
}
echo json_encode(compact('posts', 'comments'));
```

Vous pouvez faire des manipulations encore beaucoup plus complexes, ou utiliser les helpers pour formater aussi.

Note : Les classes de vue de données ne supportent pas les layouts. Elles supposent que le fichier de vue va afficher le contenu sérialisé.

---

class `XmlView`

Une classe de vue pour la génération de vue de données Xml. Voir au-dessus pour savoir comment vous pouvez utiliser XmlView dans votre application

Par défaut quand on utilise `_serialize`, XmlView va enrouler vos variables de vue sérialisées avec un noeud `<response>`. Vous pouvez définir un nom personnalisé pour ce noeud en utilisant la variable de vue `_rootNode`. Introduit dans la version 2.3 : La fonctionnalité `_rootNode` a été ajoutée.

class `JsonView`

Une classe de vue pour la génération de vue de données Json. Voir au-dessus pour savoir comment vous pouvez utiliser XmlView dans votre application.

JSONP response

Introduit dans la version 2.4. Quand vous utilisez JsonView, vous pouvez utiliser la variable de vue spéciale `_jsonp` pour permettre de retourner une réponse JSONP. La définir à `true` fait que la classe de vue vérifie si le paramètre de chaine de la requête nommé "callback" est défini et si c'est la cas, d'enrouler la réponse json dans le nom de la fonction fournie. Si vous voulez utiliser un nom personnalisé de paramètre de requête à la place de "callback", définissez `_jsonp` avec le nom requis à la place de `true`.

Helpers (Assistants)

Les Helpers (Assistants) sont des classes comme les components, pour la couche de présentation de votre application. Ils contiennent la logique de présentation qui est partagée entre plusieurs vues, éléments ou layouts. Ce chapitre vous montrera comment créer vos propres helpers et soulignera les tâches basiques que les helpers du cœur de CakePHP peuvent vous aider à accomplir.

CakePHP dispose d'un nombre de helpers qui aident à la création des vues. Ils aident à la création de balises bien-formées (y compris les formulaires), aident à la mise en forme du texte, les durées et les numéros, et peuvent même accélérer la fonctionnalité Ajax. Pour plus d'informations sur les helpers inclus dans CakePHP, allez voir *Helpers (Assistants)*.

Utiliser et configurer les Helpers

Vous activez les helpers (assistants) dans CakePHP, en faisant "prendre conscience" à un controller qu'ils existent. Chaque controller a une propriété `$helpers`, qui liste les helpers disponibles dans la vue. Pour activer un helper dans votre vue, ajoutez son nom au tableau `$helpers` du controller :

```
class BakeriesController extends AppController {
    public $helpers = array('Form', 'Html', 'Js', 'Time');
}
```

L'ajout des helpers depuis les plugins utilise la *syntaxe de plugin* utilisée partout ailleurs dans CakePHP :

```
class BakeriesController extends AppController {
    public $helpers = array('Blog.Comment');
}
```

Vous pouvez aussi ajouter les helpers depuis une action, dans ce cas, ils seront uniquement accessibles pour cette action et aucune autre dans le controller. Ceci économise de la puissance de calcul pour les autres actions qui n'utilisent pas le helper, tout en permettant de conserver le controller mieux organisé :

```
class BakeriesController extends AppController {
    public function bake {
        $this->helpers[] = 'Time';
    }
    public function mix {
        // Le Helper Time n'est pas chargé ici et n'est par conséquent
        pas disponible
    }
}
```

Si vous avez besoin d'activer un helper pour tous les controllers, ajoutez son nom dans le tableau `$helpers` du fichier `/app/Controller/AppController.php` (à créer si pas présent). N'oubliez pas d'inclure les helpers par défaut Html et Form :

```
class AppController extends Controller {
    public $helpers = array('Form', 'Html', 'Js', 'Time');
}
```

Vous pouvez passer des options dans les helpers. Ces options peuvent être utilisées pour définir les valeurs d'attributs ou modifier le comportement du helper :

```
class AwesomeHelper extends AppHelper {
    public function __construct(View $view, $settings = array()) {
        parent::__construct($view, $settings);
        debug($options);
    }
}

class AwesomeController extends AppController {
    public $helpers = array('Awesome' => array('option1' => 'valeur1'));
}
```

Depuis 2.3 les options sont fusionnées avec la propriété `Helper::$settings` du helper.

Une configuration courante est d'utiliser l'option `className`, qui vous permet de créer des helpers alias dans vos vues. Cette fonctionnalité est utile quand vous voulez remplacer `$this->Html` ou tout autre Helper de référence avec une mise en oeuvre personnalisée :

```
// app/Controller/PostsController.php
class PostsController extends AppController {
    public $helpers = array(
        'Html' => array(
            'className' => 'MyHtml'
        )
    );
}

// app/View/Helper/MyHtmlHelper.php
App::uses('HtmlHelper', 'View/Helper');
class MyHtmlHelper extends HtmlHelper {
    // Ajouter votre code pour écraser le HtmlHelper du coeur
}
```

Ce qui est au-dessus ferait un *alias* de `MyHtmlHelper` vers `$this->Html` dans vos vues.

Note : Faire un alias d'un helper remplace cette instance partout où le helper est utilisé, y compris dans les autres Helpers.

Astuce : Faire un alias des Helpers Html ou Session pendant que vous utilisez le coeur de PagesController ne fonctionnera pas. Il est préférable de copier `lib/Cake/Controller/PagesController.php` dans le dossier `app/Controller/`.

L'utilisation des configurations du helper vous permet de configurer de manière déclarative vos helpers et de garder la logique de configuration de vos actions des controllers. Si vous avez des options de configuration qui ne peuvent pas être inclues comme des parties de déclaration de classe, vous pouvez les définir dans le callback beforeRender de votre controller :

```
class PostsController extends AppController {
    public function beforeRender() {
        parent::beforeRender();
        $this->helpers['CustomStuff'] = $this->_getCustomStuffSettings();
    }
}
```

Utiliser les Helpers

Une fois que vous avez configuré les helpers que vous souhaitiez utiliser, dans votre controller, chaque helper est exposé en propriété publique dans la vue. Par exemple, si vous utilisiez `HtmlHelper`, vous seriez capable d'y accéder en faisant ce qui suit :

```
echo $this->Html->css('styles');
```

Ce qui est au-dessus appelerait la méthode `css` du HtmlHelper. Vous pouvez accéder à n'importe quel helper chargé en utilisant `$this->{$helperName}`. Il peut venir un temps où vous aurez besoin de charger dynamiquement un helper à partir d'une vue. Vous pouvez utiliser la vue du `HelperCollection` pour le faire :

```
$mediaHelper = $this->Helpers->load('Media', $mediaSettings);
```

Le HelperCollection est une *collection* et supporte l'API collection utilisée partout ailleurs dans CakePHP.

Méthodes de Callback

Les Helpers disposent de plusieurs callbacks qui vous permettent d'augmenter le processus de rendu de vue. Allez voir la documentation de *API de Helper* et *Collections* pour plus d'informations.

Créer des Helpers

Si un helper du coeur (ou l'un présenté sur github ou dans la Boulangerie) ne correspond pas à vos besoins, les helpers sont faciles à créer.

Mettons que nous voulions créer un helper, qui pourrait être utilisé pour produire un lien CSS, façonné spécialement selon vos besoins, à différents endroits de votre application. Afin de trouver une place à votre logique dans la structure de helper existante dans CakePHP, vous devrez créer une nouvelle classe dans `/app/View/Helper`. Appelons notre assistant LienHelper. Le fichier de la classe PHP devrait ressembler à quelque chose comme ceci :

```
/* /app/View/Helper/LienHelper.php */
App::uses('AppHelper', 'View/Helper');

class LienHelper extends AppHelper {
    public function lancerEdition($titre, $url) {
        // La logique pour créer le lien spécialement formaté se place
        ici...
    }
}
```

Note : Les Helpers doivent étendre soit `AppHelper` soit `Helper` ou implémenter tous les callbacks dans *API de Helper*.

Inclure d'autres Helpers

Vous souhaitez peut-être utiliser quelques fonctionnalités déjà existantes dans un autre helper. Pour faire cela, vous pouvez spécifier les helpers que vous souhaitez utiliser avec un tableau `$helpers`, formaté comme vous le feriez dans un controller :

```
/* /app/View/Helper/LienHelper.php (Utilisant d'autres helpers) */
App::uses('AppHelper', 'View/Helper');

class LienHelper extends AppHelper {
    public $helpers = array('Html');

    public function lancerEdition($titre, $url) {
```

```
        // Utilisation du helper HTML pour sortir une donnée formatée

        $link = $this->Html->link($titre, $url, array('class' => 'edit'));

        return '<div class="editOuter">' . $link . '</div>';
    }
}
```

Utiliser votre Helper

Une fois que vous avez créez votre helper et l'avez placé dans `/app/View/Helper/`, vous serez capable de l'inclure dans vos controllers en utilisant la variable spéciale `$helpers` :

```
class PostsController extends AppController {
    public $helpers = array('Lien');
}
```

Une fois que votre controller est au courant de cette nouvelle classe, vous pouvez l'utiliser dans vos vues en accédant à un objet nommé d'après le helper :

```
<!-- fait un lien en utilisant le nouveau helper -->
<?php echo $this->Lien->lancerEdition('Changer cette recette', '/recipes/edit/5'); ?>
```

Créer des fonctionnalités à vos Helpers

Tous les helpers étendent une classe spéciale, AppHelper (comme les models étendent AppModel et les controllers étendent AppController). Pour créer une fonctionnalité disponible pour tous les helpers, créez `/app/View/Helper/AppHelper.php` :

```
App::uses('Helper', 'View');

class AppHelper extends Helper {
    public function customMethod () {
    }
}
```

API de Helper

class `Helper`

La classe de base pour les Helpers. Elle fournit un nombre de méthodes utiles et des fonctionnalités pour le chargement d'autres helpers.

`Helper::webroot`(*$file*)

Décide du nom de fichier du webroot de l'application. Si un thème est actif et que le fichier existe dans le webroot du thème courant, le chemin du fichier du thème sera retourné.

`Helper::url`(*$url*, *$full = false*)

Génère une HTML escaped URL, qui délégue à `Router::url()`.

`Helper::value`(*$options = array()*, *$field = null*, *$key = 'value'*)
Récupère la valeur pour un nom d'input donné.

`Helper::domId`(*$options = null*, *$id = 'id'*)
Génère une valeur id en CamelCase pour le champ sélectionné courant. Ecraser cette méthode dans votre AppHelper vous permettra de changer la façon dont CakePHP génére les attributs ID.

Callbacks

`Helper::beforeRenderFile`(*$viewFile*)
Est appelé avant que tout fichier de vue soit rendu. Cela inclut les eléments, le vues, les vues parentes et les layouts.

`Helper::afterRenderFile`(*$viewFile*, *$content*)
Est appelé après que tout fichier de vue est rendu. Cela inclut les eléments, le vues, les vues parentes et les layouts. Un callback peut modifier et retourner `$content` pour changer la manière dont le contenu rendu est affiché dans le navigateur.

`Helper::beforeRender`(*$viewFile*)
La méthode beforeRender est appelé après la méthode beforeRender du controller, mais avant les rendus du controller de la vue et du layout Reçoit le fichier à rendre en argument.

`Helper::afterRender`(*$viewFile*)
Est appelé après que la vue est rendu, mais avant que le rendu du layout ait commencé.

`Helper::beforeLayout`(*$layoutFile*)
Est appelé avant que le rendu du layout commence. Reçoit le nom du fichier layout en argument.

`Helper::afterLayout`(*$layoutFile*)
Est appelé après que le rendu du layout est fini. Reçoit le nom du fichier layout en argument.

Helpers du coeur

CacheHelper Utilisé par le coeur pour mettre en cache le contenu de la vue.

FormHelper Créé les formulaires HTML et les éléments du formulaire qui gèrent eux-mêmes les problèmes de validation.

HTMLHelper Méthodes bien pratiques pour des balises bien formatées. Les images, les liens, les tables, les balises d'en-tête etc ....

JSHelper Utilisé pour créer du Javascript compatible avec de nombreuses librairies Javascript.

NumberHelper Formate les nombres et les monnaies.

Paginator Pagination à partir des données de Modèles et tri.

RSS Méthodes bien pratiques pour la sortie de contenu RSS et de données XML.

SessionHelper Accès pour la lecture des valeurs de session dans les vues.

TextHelper Mise en lien intelligente, mise en évidence, truchement intelligent des mots.

TimeHelper Détection de proximité (Est-ce l'année prochaine ?), formatage sympa des chaînes de caractère (Today, 10 :30 am) et conversion entre les zones de temps.

CHAPITRE 6

Models (Modèles)

Les Models sont les classes qui représentent la couche de logique dans votre application. Cela signifie qu'ils sont responsables de la gestion de presque tout ce qui concerne les données, leur validité, les interactions et l'évolution du flux d'informations dans votre domaine de travail.

Habituellement, les classes de model représentent les données et sont utilisées dans les applications CakePHP pour l'accès aux données, plus spécifiquement elles représentent une table de la base de données, mais elles ne sont pas limitées à cela et peuvent être utilisées pour accéder à tout ce qui manipule des données comme des fichiers, des services web externes, des évènements iCal ou des lignes dans un fichier CSV.

Un model peut être associé avec d'autres models. Par exemple, une Recette peut être associé avec l'Auteur de la recette ainsi qu'à l'Ingredient dans la recette.

Cette section vous expliquera quelles fonctionnalités du model peuvent être automatisées, comment outrepasser ces fonctionnalités, et quelles méthodes et propriétés un model peut avoir. Elle expliquera les différentes façons d'associer vos données. Elle décrira comment trouver, sauvegarder, et effacer des données. Au final, elle s'interessera aux sources de données.

Comprendre les Models

Un Model représente votre model de données. En programmation orientée, un model est un objet qui représente une "chose", comme une voiture, une personne ou une maison. Un blog, par exemple, peut avoir plusieurs posts et chaque post peut avoir plusieurs commentaires. Blog, Post et Commentaire sont tous des exemples de models, chacun étant associé avec un autre.

Voici un simple exemple de définition de model dans CakePHP :

```
App::uses('AppModel', 'Model');
class Ingredient extends AppModel {
    public $name = 'Ingredient';
}
```

Avec juste cette simple déclaration, le model Ingredient est doté de toutes les fonctionnalités dont vous avez besoin pour créer des requêtes, ainsi que sauvegarder et supprimer des données. Ces méthodes magiques proviennent de la classe Model de CakePHP, grâce à la magie de l'héritage. Le model Ingredient étend le model de l'application AppModel, lequel étend la classe Model interne de CakePHP. C'est cette classe Model du cœur qui fournit les fonctionnalités à l'intérieur de votre model Ingredient. `App::uses('AppModel', 'Model')` s'assure que le model est chargé sans effort pour chaque utilisation de l'instance.

La classe intermédiaire AppModel est vide et si vous n'avez pas créé la vôtre, elle provient du répertoire du coeur de cakePHP. Ecraser AppModel vous permet de définir des fonctionnalités qui doivent être rendues disponibles pour tous les models de votre application. Pour faire cela, vous avez besoin de créer votre propre fichier `AppModel.php` qui se loge dans le dossier Model, comme tous les autres models dans votre application. A la création d'un projet en utilisant *Bake*, ce fichier sera automatiquement créé pour vous.

Voir aussi *Behaviors* pour plus d'informations sur la façon d'appliquer la même logique pour de multiples models.

Retournons à notre model Ingredient, pour que cela fonctionne, créez le fichier PHP dans le repertoire `/app/Model/`. Par convention, il devra avoir le même nom que la classe ; pour cet exemple `Ingredient.php`.

Note : CakePHP créera dynamiquement un objet model pour vous si il ne peut pas trouver un fichier correspondant dans /app/Model. Cela veut également dire que si votre fichier de model n'est pas nommé correctement (ex : ingredient.php ou ingredients.php) CakePHP utilisera une instance de AppModel, plutôt que votre fichier de model "manquant" (d'un point de vue CakePHP). Si vous essayez d'utiliser une méthode que vous avez définie dans votre model ou dans un comportement attaché à votre model et que vous obtenez des erreurs SQL qui indiquent le nom de la méthode que vous appelez, c'est une indication certaine que CakePHP ne peut pas trouver votre model et que vous devez, soit vérifier les noms de fichier, soit nettoyer les fichiers temporaires ou les deux.

Note : Certains noms de classe ne sont pas utilisables pour les noms de model. Par exemple, "File" ne peut pas être utilisé puisque "File" est une classe existant déjà dans le coeur de CakePHP.

Une fois votre model défini, il peut être accédé depuis vos *Controllers*. CakePHP rend automatiquement un model disponible en accès, dès lors que son nom valide celui du controller. Par exemple, un controller nommé IngredientsController initialisera automatiquement le model Ingredient et y accédera par `$this->Ingredient` :

```
class IngredientsController extends AppController {
    public function index() {
        // Récupère tous les ingrédients et les transmet à la vue :
        $ingredients = $this->Ingredient->find('all');
        $this->set('ingredients', $ingredients);
    }
}
```

Les models associés sont accessibles à travers le model principal. Dans l'exemple suivant, Recette a une association avec le model Ingredient :

```
class Recette extends AppModel {

    public function steakRecipes() {
        $ingredient = $this->Ingredient->findByName('Steak');
        return $this->findAllByMainIngredient($ingredient['Ingredient']['id']);
    }
}
```

Cela montre comment utiliser les models qui sont déjà liés. Pour comprendre comment les associations sont définies, allez voir la *Section des associations*.

Pour en savoir plus sur les models

Associations : relier les models entre eux

Une des caractéristiques les plus puissantes de CakePHP est sa capacité d'établir les liens nécessaires entre les models d'après les informations fournies. Dans CakePHP, les liens entre models sont gérés par des associations.

Définir les relations entre différents objets à l'intérieur de votre application devrait être une tâche naturelle. Par exemple : dans une base de données de recettes, une recette peut avoir plusieurs versions, chaque version n'a qu'un seul auteur et les auteurs peuvent avoir plusieurs recettes. Le fait de définir le fonctionnement de ces relations vous permet d'accéder à vos données de manière intuitive et puissante.

Le but de cette section est de vous montrer comment concevoir, définir et utiliser les associations entre les models au sein de CakePHP.

Bien que les données peuvent être issues d'une grande variété de sources, la forme de stockage la plus répandue dans les applications web est la base de données relationnelle. La plupart de ce qui est couvert par cette section le sera dans ce contexte.

Pour des informations sur les associations avec les models de Plugin, voyez *Models du Plugin*.

Types de relations

Les quatre types d'associations dans CakePHP sont : hasOne (a un seul), hasMany (a plusieurs), belongsTo (appartient à), et hasAndBelongsToMany (HABTM) (appartient à et est composé de plusieurs).

Relation	Type d'Association	Exemple
un vers un	hasOne	Un user a un profile.
un vers plusieurs	hasMany	Un user peut avoir plusieurs recipes.
plusieurs vers un	belongsTo	Plusieurs recipes appartiennent à un user.
plusieurs vers plusieurs	hasAndBelongsToMany	Les recipes ont, et appartiennent à plusieurs ingrédients.

Les associations se définissent en créant une variable de classe nommée comme l'association que vous souhaitez définir. La variable de classe peut quelquefois se limiter à une chaîne de caractère, mais peut également être aussi complète qu'un tableau multi-dimensionnel utilisé pour définir les spécificité de l'association.

```
class User extends AppModel {
    public $hasOne = 'Profile';
    public $hasMany = array(
        'Recipe' => array(
            'className' => 'Recipe',
            'conditions' => array('Recipe.approved' => '1'),
            'order' => 'Recipe.created DESC'
        )
    );
}
```

Dans l'exemple ci-dessus, la première instance du mot 'Recipe' est ce que l'on appelle un 'Alias'. C'est un identifiant pour la relation et cela peut être ce que vous souhaitez. En règle générale, on choisit le même nom que la classe qu'il référence. Toutefois, **les alias pour chaque model doivent être uniques dans une app entière**. Par exemple, il est approprié d'avoir :

```
 class User extends AppModel {
    public $hasMany = array(
        'MyRecipe' => array(
            'className' => 'Recipe',
        )
    );
    public $hasAndBelongsToMany = array(
        'MemberOf' => array(
            'className' => 'Group',
        )
    );
}

class Group extends AppModel {
    public $hasMany = array(
        'MyRecipe' => array(
            'className' => 'Recipe',
        )
    );
    public $hasAndBelongsToMany = array(
        'Member' => array(
            'className' => 'User',
        )
    );
}
```

mais ce qui suit ne fonctionnera pas bien en toute circonstance :

```
class User extends AppModel {
    public $hasMany = array(
        'MyRecipe' => array(
            'className' => 'Recipe',
        )
    );
    public $hasAndBelongsToMany = array(
        'Member' => array(
            'className' => 'Group',
```

```
        )
    );
}

class Group extends AppModel {
    public $hasMany = array(
        'MyRecipe' => array(
            'className' => 'Recipe',
        )
    );
    public $hasAndBelongsToMany = array(
        'Member' => array(
            'className' => 'User',
        )
    );
}
```

parce que ici nous avons l'alias 'Member' se référant aux deux models User (dans Group) et Group (dans User) dans les associations HABTM. Choisir des noms non-uniques pour les alias de models à travers les models peut entraîner un comportement inattendu.

Cake va créer automatiquement des liens entre les objets model associés. Ainsi par exemple dans votre model `User`, vous pouvez accéder au model `Recipe` comme ceci :

```
$this->Recipe->someFunction();
```

De même dans votre controller, vous pouvez acceder à un model associé simplement en poursuivant les associations de votre model :

```
$this->User->Recipe->someFunction();
```

Note : Rappelez vous que les associations sont définis dans 'un sens'. Si vous définissez User hasMany Recipe, cela n'a aucun effet sur le model Recipe. Vous avez besoin de définir Recipe belongsTo User pour pouvoir accéder au model User à partir du model Recipe.

---

hasOne

Mettons en place un model User avec une relation de type hasOne vers un model Profil.

Tout d'abord, les tables de votre base de données doivent être saisies correctement. Pour qu'une relation de type hasOne fonctionne, une table doit contenir une clé étrangère qui pointe vers un enregistrement de l'autre. Dans notre cas la table profils contiendra un champ nommé user_id. Le motif de base est :

hasOne :, *l'autre* model contient la clé étrangère.

Relation	Schema
Apple hasOne Banana	bananas.apple_id
User hasOne Profile	profiles.user_id
Doctor hasOne Mentor	mentors.doctor_id

Note : Il n'est pas obligatoire de suivre les conventions de CakePHP, vous pouvez facilement outrepasser l'utilisation de toute clé étrangère dans les définitions de vos associations. Néanmoins, coller aux conventions donnera un code moins répétitif, plus facile à lire et à maintenir.

Le fichier model User sera sauvegardé dans /app/Model/User.php. Pour définir l'association 'User hasOne Profile', ajoutez la propriété $hasOne à la classe de model. Pensez à avoir un model Profile dans /app/Model/Profile.php, ou l'association ne marchera pas :

```
class User extends AppModel {
    public $hasOne = 'Profile';
}
```

Il y a deux façons de décrire cette relation dans vos fichiers de model. La méthode la plus simple est de définir l'attribut $hasOne pour une chaîne de caractère contenant le className du model associé, comme nous l'avons fait au-dessus.

Si vous avez besoin de plus de contrôle, vous pouvez définir vos associations en utilisant la syntaxe des tableaux. Par exemple, vous voudrez peut-être limiter l'association pour inclure seulement certains enregistrements.

```
class User extends AppModel {
    public $hasOne = array(
        'Profile' => array(
            'className' => 'Profile',
            'conditions' => array('Profile.published' => '1'),
            'dependent' => true
        )
    );
}
```

Les clés possibles pour les tableaux d'association incluent :

- **className** : le nom de la classe du model que l'on souhaite associer au model actuel. Si l'on souhaite définir la relation 'User a un Profile', la valeur associée à la clé 'className' devra être 'Profile'.
- **foreignKey** : le nom de la clé etrangère que l'on trouve dans l'autre model. Ceci sera particulièrement pratique si vous avez besoin de définir des relations hasOne multiples. La valeur par défaut de cette clé est le nom du model actuel (avec des underscores) suffixé avec '_id'. Dans l'exemple ci-dessus la valeur par défaut aurait été 'user_id'.
- **conditions** : un tableau des conditions compatibles de find() ou un fragment de code SQL tel que array('Profile.approved' => true)
- **fields** : une liste des champs à récupérer lorsque les données du model associé sont parcourues. Par défaut, cela retourne tous les champs.
- **order** : Un tableau des clauses order compatible de la fonction find() ou un fragment de code SQL tel que array('Profile.last_name' => 'ASC')
- **dependent** : lorsque la valeur de la clé 'dependent' est true et que la méthode delete() du model est appelée avec le paramètre 'cascade' valant true également, les enregistrements des models associés sont supprimés. Dans ce cas nous avons fixé la valeur à true de manière à ce que la suppression d'un User supprime également le Profile associé.

Une fois que cette association a été définie, les opérations de recherche sur le model User récupèrent également les enregistrements Profile liés s'il en existe :

```
//Exemple de résultats d'un appel à $this->User->find().

Array
(
    [User] => Array
        (
            [id] => 121
            [name] => Gwoo the Kungwoo
            [created] => 2007-05-01 10:31:01
        )
    [Profile] => Array
        (
            [id] => 12
            [user_id] => 121
            [skill] => Baking Cakes
            [created] => 2007-05-01 10:31:01
        )
)
```

belongsTo

Maintenant que nous avons accès aux données du Profile depuis le model User, définissons une association belongsTo (appartient a) dans le model Profile afin de pouvoir accéder aux données User liées. L'association belongsTo est un complément naturel aux associations hasOne et hasMany : elle permet de voir les données dans le sens inverse.

Lorsque vous définissez les clés de votre base de données pour une relation de type belongsTo, suivez cette convention :

belongsTo : le model *courant* contient la clé étrangère.

Relation	Schema
Banana belongsTo Apple	bananas.apple_id
Profile belongsTo User	profiles.user_id
Mentor belongsTo Doctor	mentors.doctor_id

Astuce : Si un model (table) contient une clé étrangère, elle appartient à (belongsTo) l'autre model (table).

Nous pouvons définir l'association belongsTo dans notre model Profile dans /app/Model/Profile.php en utilisant la syntaxe de chaîne de caractère comme ce qui suit :

```
class Profile extends AppModel {
    public $belongsTo = 'User';
}
```

Nous pouvons aussi définir une relation plus spécifique en utilisant une syntaxe de tableau :

```
class Profile extends AppModel {
    public $belongsTo = array(
        'User' => array(
            'className' => 'User',
```

```
            'foreignKey' => 'user_id'
        )
    );
}
```

Les clés possibles pour les tableaux d'association belongsTo incluent :

- **className** : le nom de classe du model associé au model courant. Si vous définissez une relation 'Profile belongsTo User', la clé du nom de classe devra être 'User.'
- **foreignKey** : le nom de la clé étrangère trouvée dans le model courant. C'est particulièrement pratique si vous avez besoin de définir de multiples relations belongsTo. La valeur par défaut pour cette clé est le nom au singulier de l'autre model avec des underscores, suffixé avec `_id`.
- **conditions** : un tableau de conditions compatibles find() ou de chaînes SQL comme `array('User.active' => true)`.
- **type** : le type de join à utiliser dans la requête SQL, par défaut LEFT ce qui peut ne pas correspondre à vos besoins dans toutes les situations, INNER peut être utile quand vous voulez tout de votre model principal ainsi que de vos models associés ! (Utile quand utilisé avec certaines conditions bien sur). **(NB : la valeur de type est en lettre minuscule - ex. left, inner)**
- **fields** : Une liste des champs à retourner quand les données du model associé sont récupérées. Retourne tous les champs par défaut.
- **order** : un tableau de clauses orderqui sont compatibles avec find() ou des chaînes SQL comme `array('User.username' => 'ASC')`
- **counterCache** : Si défini à true, le Model associé va automatiquement augmenter ou diminuer le champ "[singular_model_name]_count" dans la table étrangère quand vous faites un `save()` ou un `delete()`. Si c'est une chaîne alors il s'agit du nom du champ à utiliser. La valeur dans le champ counter représente le nombre de lignes liées. Vous pouvez aussi spécifier de multiples caches counter en définissant un tableau, regardez *Multiple counterCache*.
- **counterScope** : Un tableau de conditions optionnelles à utiliser pour la mise à jour du champ du cache counter.

Une fois que cette association a été définie, les opérations de find sur le model Profile va aussi récupérer un enregistrement lié de User si il existe :

```
//Exemples de résultats d'un appel de $this->Profile->find().

Array
(
   [Profile] => Array
        (
            [id] => 12
            [user_id] => 121
            [skill] => Baking Cakes
            [created] => 2007-05-01 10:31:01
        )
    [User] => Array
        (
            [id] => 121
            [name] => Gwoo the Kungwoo
            [created] => 2007-05-01 10:31:01
        )
)
```

hasMany

Prochaine étape : définir une association "User hasMany Comment". Une association hasMany nous permettra de récupérer les comments d'un user lors de la récupération d'un enregistrement User.

Lorsque vous définissez les clés de votre base de données pour une relation de type hasMany, suivez cette convention :

hasMany : l'*autre* model contient la clé étrangère.

Relation	Schema
User hasMany Comment	Comment.user_id
Cake hasMany Virtue	Virtue.cake_id
Product hasMany Option	Option.product_id

On peut définir l'association hasMany dans notre model User (/app/Model/User.php) en utilisant une chaîne de caractères de cette manière :

```
class User extends AppModel {
    public $hasMany = 'Comment';
}
```

Nous pouvons également définir une relation plus spécifique en utilisant un tableau :

```
class User extends AppModel {
    public $hasMany = array(
        'Comment' => array(
            'className' => 'Comment',
            'foreignKey' => 'user_id',
            'conditions' => array('Comment.status' => '1'),
            'order' => 'Comment.created DESC',
            'limit' => '5',
            'dependent' => true
        )
    );
}
```

Les clés possibles pour les tableaux d'association hasMany sont :

- **className** : le nom de la classe du model que l'on souhaite associer au model actuel. Si l'on souhaite définir la relation 'User hasMany Comment' (l'User a plusieurs Comments), la valeur associée à la clef 'className' devra être 'Comment'.
- **foreignKey** : le nom de la clé etrangère que l'on trouve dans l'autre model. Ceci sera particulièrement pratique si vous avez besoin de définir des relations hasMany multiples. La valeur par défaut de cette clé est le nom du model actuel (avec des underscores) suffixé avec '_id'
- **conditions** : un tableau de conditions compatibles dans find() ou des chaînes SQL comme array('Comment.visible' => true)
- **order** : un tableau de clauses order compatibles dans find() ou des chaînes SQL comme array('Profile.last_name' => 'ASC')
- **limit** : Le nombre maximum de lignes associées que vous voulez retourner.
- **offset** : Le nombre de lignes associées à enlever (étant donné les conditions et l'order courant) avant la récupération et l'association.

– **dependent** : Lorsque dependent vaut true, une suppression récursive du model est possible. Dans cet exemple, les enregistrements Comment seront supprimés lorsque leur User associé l'aura été.
– **exclusive** : Lorsque exclusive est fixé à true, la suppression récursive de model effectue la suppression avec un deleteAll() au lieu du supprimer chaque entité séparément. Cela améliore grandement la performance, mais peut ne pas être idéal dans toutes les circonstances.
– **finderQuery** : Une requête SQL complète que CakePHP peut utiliser pour retrouver les enregistrements associés au model. Ceci ne devrait être utilisé que dans les situations qui nécessitent des résultats très personnalisés. Si une de vos requêtes a besoin d'une référence à l'ID du model associé, utilisez le marqueur spécial `{$__cakeID__$}` dans la requête. Par exemple, si votre model Pomme hasMany Orange, la requête devrait ressembler à ça : `SELECT Orange.* from oranges as Orange WHERE Orange.pomme_id = {$__cakeID__$};`

Une fois que cette association a été définie, les opérations de recherche sur le model User récupèreront également les Comments reliés si ils existent :

```
//Exemple de résultats d'un appel à $this->User->find().

Array
(
    [User] => Array
        (
            [id] => 121
            [name] => Gwoo the Kungwoo
            [created] => 2007-05-01 10:31:01
        )
    [Comment] => Array
        (
            [0] => Array
                (
                    [id] => 123
                    [user_id] => 121
                    [title] => On Gwoo the Kungwoo
                    [body] => The Kungwooness is not so Gwooish
                    [created] => 2006-05-01 10:31:01
                )
            [1] => Array
                (
                    [id] => 124
                    [user_id] => 121
                    [title] => More on Gwoo
                    [body] => But what of the `Nut?
                    [created] => 2006-05-01 10:41:01
                )
        )
)
```

Une chose dont il faut se rappeler est que vous aurez besoin d'une association "Comment belongsTo User" en complément, afin de pouvoir récupérer les données dans les deux sens. Ce que nous avons défini dans cette section vous donne la possibilité d'obtenir les données de Comment depuis l'User. En ajoutant l'association "Comment belongsTo User" dans le model Comment, vous aurez la possibilité de connaître les données de l'User depuis le model Comment - cela complète la connexion entre eux et permet un flot d'informations depuis n'importe lequel des deux models.

counterCache - Cache your count()

Cette fonction vous aide à mettre en cache le count des données liées. Au lieu de compter les enregistrements manuellement via `find('count')`, le model suit lui-même tout ajout/suppression à travers le model `$hasMany` associé et augmente/diminue un champ numérique dedié dans la table du model parent.

Le nom du champ est le nom du model particulier suivi par un underscore et le mot "count" :

```
my_model_count
```

Disons que vous avez un model appelé `ImageComment` et un model appelé `Image`, vous ajouteriez un nouveau champ numérique (INT) à la table `images` et l'appeleriez `image_comment_count`.

Ici vous trouverez quelques exemples supplémentaires :

Model	Associated Model	Example
User	Image	users.image_count
Image	ImageComment	images.image_comment_count
BlogEntry	BlogEntryComment	blog_entries.blog_entry_comment_count

Une fois que vous avez ajouté le champ counter, c'est tout bon. Activez counter-cache dans votre association en ajoutant une clé `counterCache` et configurez la valeur à `true` :

```
class ImageComment extends AppModel {
    public $belongsTo = array(
        'Image' => array(
            'counterCache' => true,
        )
    );
}
```

A partir de maintenant, chaque fois que vous ajoutez ou retirez un `ImageComment` associé à `Image`, le nombre dans `image_comment_count` est ajusté automatiquement.

counterScope

Vous pouvez aussi spécifier `counterScope`. Cela vous permet de spécifier une condition simple qui dit au model quand mettre à jour (ou quand ne pas le faire, selon la façon dont on le conçoit) la valeur counter.

En utilisant notre exemple de model Image, nous pouvons le spécifier comme cela :

```
class ImageComment extends AppModel {
    public $belongsTo = array(
        'Image' => array(
            'counterCache' => true,
            'counterScope' => array('Image.active' => 1) // compte seulement si "Image" est
        )
    );
}
```

Multiple counterCache

Depuis la 2.0, CakePHP supporte les multiples `counterCache` dans une seule relation de model. Il est aussi possible de définir un `counterScope` pour chaque `counterCache`. En assumant que vous avez un model `User` et un model `Message` et que vous souhaitez être capable de compter le montant de messages lus et non lus pour chaque utilisateur.

Model	Field	Description
User	users.messages_read	Compte les `Message` lus
User	users.messages_unread	Compte les `Message` non lus
Message	messages.is_read	Determines si un `Message` est lu ou non.

Avec la configuration de votre `belongsTo` qui ressemblerait à cela :

```
class Message extends AppModel {
    public $belongsTo = array(
        'User' => array(
            'counterCache' => array(
                'messages_read' => array('Message.is_read' => 1),
                'messages_unread' => array('Message.is_read' => 0)
            )
        )
    );
}
```

hasAndBelongsToMany (HABTM)

Très bien. A ce niveau, vous pouvez déjà vous considérer comme un professionnel des associations de models CakePHP. Vous vous êtes déjà assez compétents dans les 3 types d'associations afin de pouvoir effectuer la plus grande partie des relations entre les objets.

Abordons maintenant le dernier type de relation : hasAndBelongsToMany (a et appartient à plusieurs), ou HABTM. Cette association est utilisée lorsque vous avez deux models qui ont besoin d'être reliés, de manière répétée, plusieurs fois, de plusieurs façons différentes.

La principale différence entre les relations hasMany et HABTM est que le lien entre les models n'est pas exclusif dans le cadre d'une relation HABTM. Par exemple, relions notre model Recipe avec un model Ingredient en utilisant HABTM. Le fait d'utiliser les tomates en Ingredient pour la recette de Spaghettis de ma grand-mère ne "consomme" pas l'Ingredient. Je peux aussi utiliser mes Spaghettis pour une Recipe Salade.

Les liens entre des objets liés par une association hasMany sont exclusifs. Si mon User "hasMany" Comment, un commentaire ne sera lié qu'à un user spécifique. Il ne sera plus disponible pour d'autres.

Continuons. Nous aurons besoin de mettre en place une table supplémentaire dans la base de données qui contiendra les associations HABTM. Le nom de cette nouvelle table de jointure doit inclure les noms des deux models concernés, dans l'ordre alphabétique, et séparés par un underscore (_). La table doit contenir au minimum deux champs, chacune des clés étrangères (qui devraient être des entiers) pointant sur les deux clés primaires des models concernés. Pour éviter tous problèmes, ne définissez pas une première clé composée de ces deux champs, si votre application le nécessite vous pourrez définir un index unique. Si vous prévoyez d'ajouter de quelconques informations supplémentaires à cette table, c'est une bonne idée

que d'ajouter un champ supplémentaire comme clé primaire (par convention 'id') pour rendre les actions sur la table aussi simple que pour tout autre model.

HABTM a besoin d'une table de jointure séparée qui contient les deux noms de *models*.

Relations	Champs de la table HABTM
Recipe HABTM Ingredient	**ingredients_recipes**.id, **ingredients_recipes**.ingredient_id, **ingredients_recipes**.recipe_id
Cake HABTM Fan	**cakes_fans**.id, **cakes_fans**.cake_id, **cakes_fans**.fan_id
Foo HABTM Bar	**bars_foos**.id, **bars_foos**.foo_id, **bars_foos**.bar_id

Note : Le nom des tables est par convention dans l'ordre alphabétique. Il est possible de définir un nom de table personnalisé dans la définition de l'association.

Assurez vous que les clés primaires dans les tables **cakes** et **recipes** ont un champ "id" comme assumé par convention. Si ils sont différents que ceux anticipés, il faut le changer dans la *primaryKey* du model.

Une fois que cette nouvelle table a été créée, on peut définir l'association HABTM dans les fichiers de model. Cette fois-ci, nous allons directement voir la syntaxe en tableau :

```
class Recipe extends AppModel {
    public $hasAndBelongsToMany = array(
        'Ingredient' =>
            array(
                'className' => 'Ingredient',
                'joinTable' => 'ingredients_recipes',
                'foreignKey' => 'recipe_id',
                'associationForeignKey' => 'ingredient_id',
                'unique' => true,
                'conditions' => '',
                'fields' => '',
                'order' => '',
                'limit' => '',
                'offset' => '',
                'finderQuery' => '',
                'deleteQuery' => '',
                'insertQuery' => ''
            )
    );
}
```

Les clés possibles pour un tableau définissant une association HABTM sont :

- **className** : Le nom de la classe du model que l'on souhaite associer au model actuel. Si l'on souhaite définir la relation 'Recipe HABTM Ingredient', la valeur associée à la clef 'className' devra être 'Ingredient'.
- **joinTable** : Le nom de la table de jointure utilisée dans cette association (si la table ne colle pas à la convention de nommage des tables de jointure HABTM).
- **with** : Définit le nom du model pour la table de jointure. Par défaut CakePHP créera automatiquement un model pour vous. Dans l'exemple ci-dessus la valeur aurait été RecipesTag. En utilisant cette clé vous pouvez surcharger ce nom par défaut. Le model de la table de jointure peut être utilisé comme tout autre model "classique" pour accéder directement à la table de jointure. En créant une classe model avec un tel

nom et un nom de fichier, vous pouvez ajouter tout behavior personnalisé pour les recherches de la table jointe, comme ajouter plus d'informations/colonnes à celle-ci.
- **foreignKey** : Le nom de la clé étrangère que l'on trouve dans le model actuel. Ceci sera particulièrement pratique si vous avez besoin de définir des relations HABTM multiples. La valeur par défaut de cette clé est le nom du model actuel (avec des underscores) suffixé avec '_id'.
- **associationForeignKey** : Le nom de la clé etrangère que l'on trouve dans l'autre model. Ceci sera particulièrement pratique si vous avez besoin de définir des relations HABTM multiples. La valeur par défaut de cette clé est le nom de l'autre model (avec des underscores) suffixé avec '_id'.
- **unique : Un boléen ou une chaîne de caractères `keepExisting`.**
  - Si true (valeur par défaut) Cake supprimera d'abord les enregistrements des relations existantes dans la table des clés étrangères avant d'en insérer de nouvelles, lors de la mise à jour d'un enregistrement. Ainsi les associations existantes devront être passées encore une fois lors d'une mise à jour.
  - Si false, Cake va insérer l'enregistrement lié, et aucun enregistrement joint n'est supprimé pendant une opération de sauvegarde.
  - Si `keepExisting` est défini, le behavior est similaire à *true*, mais les associations existantes ne sont pas supprimées.
- **conditions** : un tableau de conditions compatibles de find() ou des chaînes SQL. Si vous avez des conditions sur la table associée, vous devez utiliser un model 'with', et définir les associations belongsTo nécessaires sur lui.
- **fields** : Une liste des champs à récupérer lorsque les données du model associé sont parcourues. Par défaut, cela retourne tous les champs.
- **order** : un tableau de clauses order compatibles avec find() compatible ou des chaînes SQL.
- **limit** : Le nombre maximum de lignes associées que vous voulez retourner.
- **offset** : Le nombre de lignes associées à enlever (étant donnés les conditions et l'order courant) avant la récupération et l'association.
- **finderQuery, deleteQuery, insertQuery** : Une requête SQL complète que CakePHP peut utiliser pour récupérer, supprimer, ou créer des enregistrements d'un model nouvellement associé. Ceci doit être utilisé dans les situations qui nécéssitent des résultats très personnalisés.

Une fois que cette association a été définie, les opérations de recherche sur le model Recipe récupèreront également les Ingredients liés si ils existent :

```
// Exemple de résultats d'un appel a $this->Recipe->find().

Array
(
    [Recipe] => Array
        (
            [id] => 2745
            [name] => Chocolate Frosted Sugar Bombs
            [created] => 2007-05-01 10:31:01
            [user_id] => 2346
        )
    [Ingredient] => Array
        (
            [0] => Array
                (
                    [id] => 123
                    [name] => Chocolate
                )
```

```
            [1] => Array
                 (
                     [id] => 124
                     [name] => Sugar
                 )
            [2] => Array
                 (
                     [id] => 125
                     [name] => Bombs
                 )
        )
)
```

N'oubliez pas de définir une association HABTM dans le model Ingredient si vous souhaitez retrouver les données de Recipe lorsque vous manipulez le model Ingredient.

Note : Les données HABTM sont traitées comme un ensemble complet, chaque fois qu'une nouvelle association de donnée est ajoutée, l'ensemble complet de lignes associées dans la base de données est enlevé et recrée ainsi vous devrez toujours passer l'ensemble des données définies pour sauvegarder. Pour avoir une alternative à l'utilisation de HABTM, regardez *hasMany through (Le Model Join)*

Astuce : Pour plus d'informations sur la sauvegarde des objets HABTM regardez *Sauvegarder les Données de Model Lié (HABTM=HasAndBelongsToMany)*

---

hasMany through (Le Model Join)

Il est parfois nécéssaire de stocker des données supplémentaires avec une association many to many. Considérons ce qui suit

Student hasAndBelongsToMany Course

Course hasAndBelongsToMany Student

En d'autres termes, un Student peut avoir plusieurs (many) Courses et un Course peut être pris par plusieurs (many) Students. C'est une association simple de many to many nécéssitant une table comme ceci :

```
id | student_id | course_id
```

Maintenant si nous souhaitions stocker le nombre de jours que les students doivent faire pour leur course et leur grade final ? La table que nous souhaiterions serait comme ceci :

```
id | student_id | course_id | days_attended | grade
```

Le problème est que hasAndBelongsToMany ne vas pas supporter ce type de scénario parce que quand les associations hasAndBelongsToMany sont sauvegardées, l'association est d'abord supprimée. Vous perderiez les données supplémentaires dans les colonnes qui ne seraient pas remplacées dans le nouvel ajout.

Modifié dans la version 2.1. Vous pouvez définir la configuration de `unique` à `keepExisting` contournant la perte de données supplémentaires pendant l'opération de sauvegarde. Regardez la clé `unique` dans *HABTM association arrays*.

La façon d'implémenter nos exigences est d'utiliser un **join model**, autrement connu comme une association **hasMany through**. Cela étant fait, l'association est elle-même un model. Ainsi, vous pouvez créer un nouveau model CourseMembership. Regardez les models suivants. :

```
// Student.php
class Student extends AppModel {
    public $hasMany = array(
        'CourseMembership'
    );
}

// Course.php

class Course extends AppModel {
    public $hasMany = array(
        'CourseMembership'
    );
}

// CourseMembership.php

class CourseMembership extends AppModel {
    public $belongsTo = array(
        'Student', 'Course'
    );
}
```

Le model join CourseMembership identifie de façon unique une participation d'un Student à un Course en plus d'ajouter des meta-informations.

Les models Join sont des choses particulièrement pratiques à utiliser et Cake facilite cela avec les associations intégrées hasMany and belongsTo et la fonctionnalité de saveAll.

Créer et Détruire des Associations à la Volée

Quelquefois il devient nécessaire de créer et détruire les associations de models à la volée. Cela peut être le cas pour un certain nombre de raisons :

- Vous voulez réduire la quantité de données associées qui seront récupérées, mais toutes vos associations sont sur le premier niveau de récursion.
- Vous voulez changer la manière dont une association est définie afin de classer ou filtrer les données associées.

La création et destruction de ces associations se font en utilisant les méthodes de models CakePHP bindModel() et unbindModel(). (Il existe aussi un behavior très utile appelé "Containable", merci de vous référer à la section du manuel sur les behaviors intégrés pour plus d'informations). Mettons en place quelques models pour pouvoir ensuite voir comment fonctionnent bindModel() et unbindModel(). Nous commencerons avec deux models :

```
class Leader extends AppModel {
    public $hasMany = array(
        'Follower' => array(
            'className' => 'Follower',
            'order' => 'Follower.rank'
        )
    );
}

class Follower extends AppModel {
    public $name = 'Follower';
}
```

Maintenant, dans le controller LeadersController, nous pouvons utiliser la méthode find() du model Leader pour retrouver un Leader et les Suiveurs associés. Comme vous pouvez le voir ci-dessus, le tableau d'association dans le model Leader définit une relation "Leader hasMany (a plusieurs) Suiveurs". Dans un but démonstratif, utilisons unbindModel() pour supprimer cette association dans une action du controller :

```
public function some_action() {
    // Ceci récupère tous les Leaders, ainsi que leurs Suiveurs
    $this->Leader->find('all');

    // Supprimons la relation hasMany() ...
    $this->Leader->unbindModel(
        array('hasMany' => array('Follower'))
    );

    // Désormais l'utilisation de la fonction find() retournera
    // des Leaders, sans aucun Suiveurs
    $this->Leader->find('all');

    // NOTE : unbindModel n'affecte que la prochaine fonction find.
    // Un autre appel à find() utilisera les informations d'association
    // telles que configurée.

    // Nous avons déjà utilisé findAll('all') après unbindModel(),
    // ainsi cette ligne récupèrera une fois encore les Leaders
    // avec leurs Suiveurs ...
    $this->Leader->find('all');
}
```

Note : Enlever ou ajouter des associations en utilisant bindModel() et unbindModel() ne fonctionne que pour la *prochaine* opération sur le model, à moins que le second paramètre n'ait été fixé à false. Si le second paramètre a été fixé à *false*, le lien reste en place pour la suite de la requête.

Voici un exemple basique d'utilisation de unbindModel() :

```
$this->Model->unbindModel(
    array('associationType' => array('associatedModelClassName'))
);
```

Maintenant que nous sommes arrivés à supprimer une association à la volée, ajoutons-en une. Notre Leader jusqu'à présent sans Principes a besoin d'être associé à quelques Principes. Le fichier de model pour notre model Principe est dépouillé, il n'y a que la ligne var $name. Associons à la volée des Principes à notre Leader (mais rappelons-le, seulement pour la prochaine opération find). Cette fonction apparaît dans le controller LeadersController :

```
public function another_action() {
    // Il n'y a pas d'association Leader hasMany Principe
    // dans le fichier de model Meneur.php, ainsi un find
    // situé ici ne récupèrera que les Leaders.
    $this->Leader->find('all');

    // Utilisons bindModel() pour ajouter une nouvelle association
    // au model Leader :
    $this->Leader->bindModel(
        array('hasMany' => array(
                'Principle' => array(
                    'className' => 'Principle'
                )
            )
        )
    );

    // Maintenant que nous les avons associés correctement,
    // nous pouvons utiliser la fonction find une seule fois
    // pour récupérer les Leaders avec leurs Principes associés :
    $this->Leader->find('all');
}
```

Ça y est, vous y êtes. L'utilisation basique de bindModel() est l'encapsulation d'un tableau d'association classique, dans un tableau dont la clé est le nom du type d'association que vous essayez de créer :

```
$this->Model->bindModel(
    array('associationName' => array(
            'associatedModelClassName' => array(
                // les clés d'association normale vont ici...
            )
        )
    )
);
```

Bien que le model nouvellement associé n'ait besoin d'aucune définition d'association dans son fichier de model, il devra tout de même contenir les clés afin que la nouvelle association fonctionne bien.

Plusieurs relations avec le même model

Il y a des cas où un Model a plus d'une relation avec un autre Model. Par exemple, vous pourriez avoir un model Message qui a deux relations avec le model User. Une relation avec l'user qui envoie un message et une seconde avec l'user qui reçoit le message. La table messages aura un champ user_id, mais aussi un champ receveur_id. Maintenant, votre model Message peut ressembler à quelque chose comme :

```
class Message extends AppModel {
    public $belongsTo = array(
        'Sender' => array(
            'className' => 'User',
            'foreignKey' => 'user_id'
        ),
        'Recipient' => array(
            'className' => 'User',
            'foreignKey' => 'recipient_id'
        )
    );
}
```

Recipient est un alias pour le model User. Maintenant, voyons à quoi devrait ressembler le model User :

```
class User extends AppModel {
    public $hasMany = array(
        'MessageSent' => array(
            'className' => 'Message',
            'foreignKey' => 'user_id'
        ),
        'MessageReceived' => array(
            'className' => 'Message',
            'foreignKey' => 'recipient_id'
        )
    );
}
```

Il est aussi possible de créer des associations sur soi-même comme montré ci-dessous :

```
class Post extends AppModel {
    public $name = 'Post';

    public $belongsTo = array(
        'Parent' => array(
            'className' => 'Post',
            'foreignKey' => 'parent_id'
        )
    );

    public $hasMany = array(
        'Children' => array(
            'className' => 'Post',
            'foreignKey' => 'parent_id'
        )
    );
}
```

Récupérer un tableau imbriqué d'enregitrements associés :

Si votre table a un champ `parent_id`, vous pouvez aussi utiliser *find('threaded')* pour récupérer un tableau imbriqué d'enregistrements en utilisant une seule requête sans définir aucune association.

Tables jointes

En SQL, vous pouvez combiner des tables liées en utilisant la clause JOIN. Ceci vous permet de réaliser des recherches complexes à travers des tables multiples (par ex. : rechercher les posts selon plusieurs tags donnés).

Dans CakePHP, certaines associations (belongsTo et hasOne) effectuent des jointures automatiques pour récupérer les données, vous pouvez donc lancer des requêtes pour récupérer les models basés sur les données de celui qui est lié.

Mais ce n'est pas le cas avec les associations hasMany et hasAndBelongsToMany. C'est là que les jointures forcées viennent à notre secours. Vous devez seulement définir les jointures nécessaires pour combiner les tables et obtenir les résultats désirés pour votre requête.

Note : Souvenez-vous que vous avez besoin de définir la récursivité à -1 pour que cela fonctionne. Par exemple : $this->Channel->recursive = -1 ;

Pour forcer une jointure entre tables, vous avez besoin d'utiliser la syntaxe "moderne" de Model : :find(), en ajoutant une clé 'joins' au tableau $options. Par exemple :

```
$options['joins'] = array(
    array('table' => 'channels',
        'alias' => 'Channel',
        'type' => 'LEFT',
        'conditions' => array(
            'Channel.id = Item.channel_id',
        )
    )
);

$Item->find('all', $options);
```

Note : Notez que les tableaux 'join' ne sont pas indexés.

Dans l'exemple ci-dessus, un model appelé Item est joint à gauche à la table channels. Vous pouvez ajouter un alias à la table, avec le nom du Model, ainsi les données retournées se conformeront à la structure de données de CakePHP.
- **table** : La table pour la jointure.
- **alias** : un alias vers la table. Le nom du model associé avec la table est le meilleur choix.
- **type** : Le type de jointure : inner, left ou right.
- **conditions** : Les conditions pour réaliser la jointure.

Avec joins, vous pourriez ajouter des conditions basées sur les champs du model relié :

```
$options['joins'] = array(
    array('table' => 'channels',
        'alias' => 'Channel',
        'type' => 'LEFT',
        'conditions' => array(
            'Channel.id = Item.channel_id',
```

```
        )
    )
);

$options['conditions'] = array(
    'Channel.private' => 1
);

$privateItems = $Item->find('all', $options);
```

Au besoin, vous pourriez réaliser plusieurs jointures dans une hasAndBelongsToMany :

Supposez une association Book hasAndBelongsToMany Tag. Cette relation utilise une table books_tags comme table de jointure, donc vous avez besoin de joindre la table books à la table books_tags et celle-ci avec la table tags :

```
$options['joins'] = array(
    array('table' => 'books_tags',
        'alias' => 'BooksTag',
        'type' => 'inner',
        'conditions' => array(
            'Books.id = BooksTag.books_id'
        )
    ),
    array('table' => 'tags',
        'alias' => 'Tag',
        'type' => 'inner',
        'conditions' => array(
            'BooksTag.tag_id = Tag.id'
        )
    )
);

$options['conditions'] = array(
    'Tag.tag' => 'Novel'
);

$books = $Book->find('all', $options);
```

Utiliser joins vous permet d'avoir un maximum de flexibilité dans la façon dont CakePHP gère les associations et récupère les données, cependant dans la plupart des cas, vous pouvez utiliser d'autres outils pour arriver aux mêmes résultats comme de définir correctement les associations, lier les models à la volée et utiliser le behavior Containable. Cette fonctionnalité doit être utilisée avec attention car elle peut conduire, dans certains cas, à quelques erreurs SQL lorsqu'elle est combinée à d'autres techniques décrites précédemment pour les models associés.

Récupérer vos données

Comme mentionné avant, un des rôles de la couche Model est d'obtenir les données à partir de multiples types de stockage. La classe Model de CakePHP est livrée avec quelques fonctions qui vous aident à chercher

ces données, à les trier, les paginer, et les filtrer. La fonction la plus courante que vous utiliserez dans les models est `Model::find()`.

find

`find(string $type = 'first', array $params = array())`

Find est, parmi toutes les fonctions de récupération de données des models, une véritable bête de somme multi-fonctionnelle. `$type` peut être `'all'`, `'first'`, `'count'`, `'list'`, `'neighbors'`, `'threaded'`, ou tout autre fonction de recherche que vous définissez. Gardez à l'esprit que `$type` est sensible à la casse. Utiliser un caractère majuscule (par exemple `All`) ne produira pas le résultat attendu.

`$params` est utilisé pour passer tous les paramètres aux différentes formes de find et il a les clés suivantes disponibles par défaut - qui sont toutes optionnelles :

```
array(
    'conditions' => array('Model.field' => $cetteValeur), //tableau de conditions
    'recursive' => 1, //int
    'fields' => array('Model.champ1', 'DISTINCT Model.champ2'), //tableau de champs nommés
    'order' => array('Model.created', 'Model.champ3 DESC'), //chaîne de caractère ou table
    'group' => array('Model.champ'), //champs en GROUP BY
    'limit' => n, //int
    'page' => n, //int
    'offset' => n, //int
    'callbacks' => true //autres valeurs possible sont false, 'before', 'after'
)
```

Il est possible également, d'ajouter et d'utiliser d'autres paramètres, dont il est fait usage dans quelques types de find, dans des behaviors (comportements) et, bien sûr, dans vos propres méthodes de model.

find('first')

`find('first', $params)` retournera un résultat, vous devriez utiliser ceci dans tous les cas où vous attendez un seul résultat. Ci-dessous, quelques exemples simples (code du controller) :

```
public function une_fonction() {
    // ...
    $articleADemiAleatoire = $this->Article->find('first');
    $dernierCree = $this->Article->find('first', array(
        'order' => array('Article.created' => 'desc')
    ));
    $specifiquementCeluiCi = $this->Article->find('first', array(
        'conditions' => array('Article.id' => 1)
    ));
    // ...
}
```

Dans le premier exemple, aucun paramètre n'est passé au find ; par conséquent aucune condition ou ordre de tri ne seront utilisés. Le format retourné par un appel à `find('first')` est de la forme :

```
Array
(
    [NomDuModel] => Array
        (
            [id] => 83
            [champ1] => valeur1
            [champ2] => valeur2
            [champ3] => valeur3
        )

    [NomDuModelAssocie] => Array
        (
            [id] => 1
            [champ1] => valeur1
            [champ2] => valeur2
            [champ3] => valeur3
        )
)
```

find('count')

`find('count', $params)` retourne une valeur de type entier. Ci-dessous, quelques exemples simples (code du controller) :

```
public function une_fonction() {
    // ...
    $total = $this->Article->find('count');
    $en_attente = $this->Article->find('count', array(
        'conditions' => array('Article.status' => 'pending')
    ));
    $authors = $this->Article->User->find('count');
    $auteursPublies = $this->Article->find('count', array(
       'fields' => 'DISTINCT Article.user_id',
       'conditions' => array('Article.status !=' => 'pending')
    ));
    // ...
}
```

Note : Ne passez pas `fields` comme un tableau à `find('count')`. Vous devriez avoir besoin de spécifier seulement des champs pour un count DISTINCT (parce que sinon, le décompte est toujours le même - il est imposé par les conditions).

find('all')

`find('all', $params)` retourne un tableau de résultats (potentiellement multiples). C'est en fait le mécanisme utilisé par toutes les variantes de `find()`, ainsi que par `paginate`. Ci-dessous, quelques exemples simples (code du controller) :

```
public function une_fonction() {
    // ...
    $tousLesArticles = $this->Article->find('all');
    $en_attente = $this->Article->find('all', array(
        'conditions' => array('Article.status' => 'pending')
    ));
    $tousLesAuteurs = $this->Article->User->find('all');
    $tousLesAuteursPublies = $this->Article->User->find('all', array(
        'conditions' => array('Article.status !=' => 'pending')
    ));
    // ...
}
```

Note : Dans l'exemple ci-dessus `$tousLesAuteurs` contiendra chaque user de la table users, il n'y aura pas de condition appliquée à la recherche puisqu'aucune n'a été passée.

Les résultats d'un appel à `find('all')` seront de la forme suivante :

```
Array
(
    [0] => Array
        (
            [NomDuModel] => Array
                (
                    [id] => 83
                    [champ1] => valeur1
                    [champ2] => valeur2
                    [champ3] => valeur3
                )

            [NomDuModelAssocie] => Array
                (
                    [id] => 1
                    [champ1] => valeur1
                    [champ2] => valeur2
                    [champ3] => valeur3
                )

        )
)
```

find('list')

`find('list', $params)` retourne un tableau indexé, pratique pour tous les cas où vous voudriez une liste telle que celles remplissant les champs select. Ci-dessous, une paire d'exemples simples (code du controller) :

```
public function une_function() {
    // ...
    $tousLesArticles = $this->Article->find('list');
```

```
    $en_attente = $this->Article->find('list', array(
        'conditions' => array('Article.status' => 'pending')
    ));
    $tousLesAuteurs = $this->Article->User->find('list');
    $tousLesAuteursPublies = $this->Article->find('list', array(
        'fields' => array('User.id', 'User.name'),
        'conditions' => array('Article.status !=' => 'pending'),
        'recursive' => 0
    ));
    // ...
}
```

Note : Dans l'exemple ci-dessus `$tousLesAuteurs` contiendra chaque user de la table users, il n'y aura pas de condition appliquée à la recherche puisqu'aucune n'a été passée.

Le résultat d'un appel à `find('list')` sera de la forme suivante :

```
Array
(
    //[id] => 'valeurAffichage',
    [1] => 'valeurAffichage1',
    [2] => 'valeurAffichage2',
    [4] => 'valeurAffichage4',
    [5] => 'valeurAffichage5',
    [6] => 'valeurAffichage6',
    [3] => 'valeurAffichage3',
)
```

En appelant `find('list')`, les champs (`fields`) passés sont utilisés pour déterminer ce qui devrait être utilisé comme clé, valeur du tableau et, optionnellement, par quoi regrouper les résultats (group by). Par défaut la clé primaire du model est utilisé comme clé et le champ affiché (display field qui peut être configuré en utilisant l'attribut *displayField* du model) est utilisé pour la valeur. Quelques exemples complémentaires pour clarifier les choses :

```
public function une_function() {
    // ...
    $juste_les_usernames = $this->Article->User->find('list', array(
        'fields' => array('User.username')
    ));
    $correspondanceUsername = $this->Article->User->find('list', array(
        'fields' => array('User.username', 'User.first_name')
    ));
    $groupesUsername = $this->Article->User->find('list', array(
        'fields' => array('User.username', 'User.first_name', 'User.group')
    ));
    // ...
}
```

Avec l'exemple de code ci-dessus, les variables résultantes devraient ressembler à quelque chose comme cela :

```
$juste_les_usernames = Array
(
    //[id] => 'username',
    [213] => 'AD7six',
    [25] => '_psychic_',
    [1] => 'PHPNut',
    [2] => 'gwoo',
    [400] => 'jperras',
)

$correspondanceUsername = Array
(
    //[username] => 'firstname',
    ['AD7six'] => 'Andy',
    ['_psychic_'] => 'John',
    ['PHPNut'] => 'Larry',
    ['gwoo'] => 'Gwoo',
    ['jperras'] => 'Joël',
)

$groupesUsername = Array
(
    ['Utilisateur'] => Array
    (
        ['PHPNut'] => 'Larry',
        ['gwoo'] => 'Gwoo',
    )

    ['Admin'] => Array
    (
        ['_psychic_'] => 'John',
        ['AD7six'] => 'Andy',
        ['jperras'] => 'Joël',
    )

)
```

find('threaded')

`find('threaded', $params)` retourne un tableau imbriqué et est particulièrement approprié si vous voulez utiliser le champ `parent_id` des données de votre model, pour construire les résultats associés. Ci-dessous, quelques exemples simples (code du controller) :

```
public function une_function() {
    // ...
    $toutesLesCategories = $this->Category->find('threaded');
    $quelquesCategories = $this->Comment->find('threaded', array(
        'conditions' => array('article_id' => 50)
    ));
    // ...
}
```

Astuce : Un meilleur moyen de gérer les données imbriquées est d'utiliser le behavior *Tree*

Dans l'exemple ci-dessus, `$toutesLesCategories` contiendra un tableau imbriqué représentant la structure entière de categorie. Le résultat d'un appel à `find('threaded')` sera de la forme suivante :

```
Array
(
    [0] => Array
    (
        [NomDuModel] => Array
        (
            [id] => 83
            [parent_id] => null
            [champ1] => valeur1
            [champ2] => valeur2
            [champ3] => valeur3
        )

        [NomDuModelAssocie] => Array
        (
            [id] => 1
            [champ1] => valeur1
            [champ2] => valeur2
            [champ3] => valeur3
        )

        [children] => Array
        (
            [0] => Array
            (
                [NomDuModel] => Array
                (
                    [id] => 42
                    [parent_id] => 83
                    [champ1] => valeur1
                    [champ2] => valeur2
                    [champ3] => valeur3
                )

                [NomDuModelAssocie] => Array
                (
                    [id] => 2
                    [champ1] => valeur1
                    [champ2] => valeur2
                    [champ3] => valeur3
                )

                [children] => Array
                (
                )
            )
            ...
```

```
        )
    )
)
```

L'ordre dans lequel les résultats apparaissent peut être modifié, puisqu'il est influencé par l'ordre d'exécution. Par exemple, si `'order' => 'name ASC'` est passé dans les paramètres de `find('threaded')`, les résultats apparaîtront ordonnés par nom. De même que tout ordre peut être utilisé, il n'y a pas de condition intrinsèque à cette méthode pour que le meilleur résultat soit retourné en premier.

Warning : Si vous spécifiez `fields`, vous aurez besoin de toujours inclure le parent_id (ou son alias courant) :

```
public function some_function() {
    $categories = $this->Category->find('threaded', array(
        'fields' => array('id', 'name', 'parent_id')
    ));
}
```

Sinon le tableau retourné ne sera pas de la structure imbriquée attendue du dessus.

find('neighbors')

`find('neighbors', $params)` exécutera un find similaire à 'first', mais retournera les lignes précédentes et suivantes à celle que vous requêtez. Ci-dessous, un exemple simple (code du controller) :

```
public function une_function() {
   $neighbors = $this->Article->find('neighbors', array('field' => 'id', 'value' => 3));
}
```

Vous pouvez voir dans cet exemple, les deux éléments requis par le tableau `$params` : field et value. Les autres éléments sont toujours autorisés, comme dans tout autre find (Ex : si votre model agit comme un containable, alors vous pouvez spécifier 'contain' dans `$params`). Le format retourné par un appel à `find('neighbors')` est de la forme :

```
Array
(
    [prev] => Array
    (
        [NomDuModel] => Array
        (
            [id] => 2
            [champ1] => valeur1
            [champ2] => valeur2
            ...
        )
        [NomDuModelAssocie] => Array
        (
            [id] => 151
            [champ1] => valeur1
```

```
            [champ2] => valeur2
            ...
        )
    )
    [next] => Array
    (
        [NomDuModel] => Array
        (
            [id] => 4
            [champ1] => valeur1
            [champ2] => valeur2
            ...
        )
        [NomDuModelAssocie] => Array
        (
            [id] => 122
            [champ1] => valeur1
            [champ2] => valeur2
            ...
        )
    )
)
```

Note : Notez que le résultat contient toujours seulement deux éléments de premier niveau : prev et next. Cette fonction ne possède pas de variable récursive par défaut d'un model. Le paramètre récursif doit être passé dans les paramètres de chaque appel.

Créer des types de recherche personnalisés

La méthode `find` est assez flexible pour accepter vos recherches personnalisées, ceci est fait en déclarant vos propres types dans une variable de model et en intégrant une fonction spéciale dans votre classe de model.

Un type de recherche Model est un raccourci pour les options de recherche. Par exemple, les deux finds suivants sont équivalents

```
$this->User->find('first');
$this->User->find('all', array('limit' => 1));
```

Ci-dessous les différents types de find du coeur :
- `first`
- `all`
- `count`
- `list`
- `threaded`
- `neighbors`

Mais qu'en est-il des autres types ? Mettons que vous souhaitiez un finder pour tous les articles publiés dans votre base de données. Le premier changement que vous devez faire est d'ajouter votre type dans la variable `Model::$findMethods` dans le model

```
class Article extends AppModel {
    public $findMethods = array('available' =>  true);
}
```

Au fond, cela dit juste à CakePHP d'accepter la valeur `available` pour premier argument de la fonction `find`. Prochaine étape est l'intégration de la fonction `_findAvailable`. Cela est fait par convention, si vous voulez intégrer un finder appelé `maSuperRecherche` ensuite la méthode à intégrer s'appellera `_findMaSuperRecherche`.

```
class Article extends AppModel {
    public $findMethods = array('available' =>  true);

    protected function _findAvailable($state, $query, $results = array()) {
        if ($state === 'before') {
            $query['conditions']['Article.publie'] = true;
            return $query;
        }
        return $results;
    }
}
```

Cela vient avec l'exemple suivant (code du controller) :

```
class ArticlesController extends AppController {

    // Trouvera tous les articles publiés et les ordonne en fonction de la colonne created
    public function index() {
        $articles = $this->Article->find('available', array(
            'order' => array('created' => 'desc')
        ));
    }

}
```

Les méthodes spéciales `_find[Type]` reçoivent 3 arguments comme montré ci-dessus. Le premier signifie que l'état de l'exécution de la requête, qui peut être soit `before` ou `after`. Cela est fait de cette façon parce que cette fonction est juste une sorte de fonction callback qui a la capacité de modifier la requête avant qu'elle se fasse, ou de modifier les résultats après qu'ils sont récupérés.

Typiquement, la première chose à vérifier dans notre fonction find est l'état de la requête. L'état `before` est le moment de modifier la requête, de former les nouvelles associations, d'appliquer plus de behaviors, et d'interpreter toute clé spéciale qui est passé dans le deuxième argument de `find`. Cet état nécessite que vous retourniez l'argument $query (modifié ou non).

L'état `after` est l'endroit parfait pour inspecter les résultats, injecter de nouvelles données, le traiter pour retourner dans un autre format, ou faire ce que vous voulez sur les données fraichement récupérées. Cet état nécessite que vous retourniez le tableau $results (modifié ou non).

Vous pouvez créer autant de finders personnalisés que vous souhaitez, et ils sont une bonne façon de réutiliser du code dans votre application à travers les models.

Il est aussi possible de paginer grâce à un type de find personnalisé comme suit :

```
class ArticlesController extends AppController {

    // Va paginer tous les articles publiés
    public function index() {
        $this->paginate = array('available');
        $articles = $this->paginate();
        $this->set(compact('articles'));
    }

}
```

Configurer la propriété `$this->paginate` comme ci-dessus dans le controller fera que le `type` de find deviendra `available`, et vous permettra aussi de continuer à modifier les résultats trouvés.

Si le compte de votre page de pagination devient fausse, il peut être nécessaire d'ajouter le code suivant à votre `AppModel`, ce qui devrait régler le compte de pagination :

```
class AppModel extends Model {

/**
 * Retire la clé 'fields' du compte de la requête find personnalisée
 * quand c'est un tableau, comme il cassera entièrement l'appel
 * Model::_findCount() call
 *
 * @param string $state Soit "before" soit "after"
 * @param array $query
 * @param array $results
 * @return int Le nombre d'enregistrements trouvés, ou false
 * @access protected
 * @see Model::find()
 */
    protected function _findCount($state, $query, $results = array()) {
        if ($state === 'before') {
            if (isset($query['type']) && isset($this->findMethods[$query['type']])) {
                $query = $this->{'_find' . ucfirst($query['type'])}('before', $query);
                if (!empty($query['fields']) && is_array($query['fields'])) {
                    if (!preg_match('/^count/i', current($query['fields']))) {
                        unset($query['fields']);
                    }
                }
            }
        }
        return parent::_findCount($state, $query, $results);
    }

}
```

Modifié dans la version 2.2. Vous n'avez plus besoin de surcharger _findCount pour régler les problèmes des count de résultat incorrects. L'état `'before'` de vos finder personnalisés vous permettent maintenant d'être appelés à nouveaux avec $query['operation'] = 'count'. Le $query retourné va être utilisé dans `_findCount()`. Si nécéssaire, vous pouvez distinguer en vérifiant pour la clé `'operation'` et retourner un `$query` différent :

```
protected function _findAvailable($state, $query, $results = array()) {
    if ($state === 'before') {
        $query['conditions']['Article.published'] = true;
        if (!empty($query['operation']) && $query['operation'] === 'count') {
            return $query;
        }
        $query['joins'] = array(
            //array of required joins
        );
        return $query;
    }
    return $results;
}
```

Types Magiques de Recherche

Ces fonctions magiques peuvent être utilisées comme un raccourci pour rechercher dans vos tables sur un champ précis. Ajoutez simplement le nom du champ (au format CamelCase) à la fin de ces fonctions et fournissez le critère de recherche pour ce champ comme premier paramètre.

Les fonctions findAllBy() retourneront des résultats dans un format comme `find('all')`, alors que findBy() retourne dans le même format que `find('first')`

findAllBy

```
findAllBy<fieldName>(string $value, array $fields, array $order, int
$limit, int $page, int $recursive)
```

findAllBy<x> Exemple	Corresponding SQL Fragment
`$this->Product->findAllByOrderStatus('3');`	`Product.order_status = 3`
`$this->Recipe->findAllByType('Cookie');`	`Recipe.type = 'Cookie'`
`$this->User->findAllByLastName('Anderson');`	`User.last_name = 'Anderson'`
`$this->Cake->findAllById(7);`	`Cake.id = 7`
`$this->User->findAllByEmailOrUsername('jhon', 'jhon');`	`User.email = 'jhon' OR User.username = 'jhon';`
`$this->User->findAllByUsernameAndPassword('jhon', '123');`	`User.username = 'jhon' AND User.password = '123';`
`$this->User->findAllByLastName('psychic', array(), array('User.user_name => 'asc'));`	`User.last_name = 'psychic' ORDER BY User.user_name ASC`

Le résultat retourné est un tableau formaté un peu comme ce que donnerait `find('all')`.

findBy

```
findBy<fieldName>(string $value);
```

Les fonctions magiques findBy acceptent aussi quelques paramètres optionnels :

`findBy<fieldName>(string $value[, mixed $fields[, mixed $order]]);`

findBy<x> Exemple	Corresponding SQL Fragment
`$this->Produit->findByOrderStatus('3');`	`Product.order_status = 3`
`$this->Recipe->findByType('Cookie');`	`Recipe.type = 'Cookie'`
`$this->User->findByLastName('Anderson');`	`User.last_name = 'Anderson';`
`$this->User->findByEmailOrUsername('jhon', 'jhon');`	`User.email = 'jhon' OR User.username = 'jhon';`
`$this->User->findByUsernameAndPassword('jhon', '123');`	`User.username = 'jhon' AND User.password = '123';`
`$this->Cake->findById(7);`	`Cake.id = 7`

Les fonctions findBy() retournent des résultats comme `find('first')`.

`Model::query()`

`query(string $query)`

Les appels SQL que vous ne pouvez pas ou ne voulez pas faire grâce aux autres méthodes de model (attention, il y a très peu de circonstances où cela se vérifie), peuvent être exécutés en utilisant la méthode `query()`.

Si vous utilisez souvent cette méthode dans votre application, assurez-vous de connaître la librairie *Data Sanitization (Nettoyage des Données)* de CakePHP, qui vous aide à nettoyer les données provenant des users, des attaques par injection et cross-site scripting.

Note : `query()` ne respecte pas $Model->cacheQueries car cette fonctionnalité est par nature déconnectée de tout ce qui concerne l'appel du model. Pour éviter les appels au cache de requêtes, fournissez un second argument false, par exemple : `query($query, $cachequeries = false)`.

`query()` utilise le nom de la table déclarée dans la requête comme clé du tableau de données retourné, plutôt que le nom du model. Par exemple :

```
$this->Picture->query("SELECT * FROM pictures LIMIT 2;");
```

pourrait retourner :

```
Array
(
    [0] => Array
    (
        [pictures] => Array
        (
            [id] => 1304
            [user_id] => 759
        )
    )

    [1] => Array
```

```
    (
        [pictures] => Array
        (
            [id] => 1305
            [user_id] => 759
        )
    )
)
```

Pour utiliser le nom du model comme clé du tableau et obtenir un résultat cohérent avec ce qui est retourné par les méthodes Find, la requête doit être réécrite :

```
$this->Picture->query("SELECT * FROM pictures AS Picture LIMIT 2;");
```

ce qui retourne :

```
Array
(
    [0] => Array
    (
        [Picture] => Array
        (
            [id] => 1304
            [user_id] => 759
        )
    )

    [1] => Array
    (
        [Picture] => Array
        (
            [id] => 1305
            [user_id] => 759
        )
    )
)
```

Note : Cette syntaxe et la structure de tableau correspondante est valide seulement pour MySQL. Cake ne fournit pas de données d'abstraction quand les requêtes sont lancées manuellement, donc les résultats exacts vont varier entre les bases de données.

`Model::field()`

`field(string $name, array $conditions = null, string $order = null)`

Retourne la valeur d'un champ unique, spécifié par `$name`, du premier enregistrement correspondant aux $conditions ordonnées par $order. Si aucune condition n'est passée et que l'id du model est fixé, cela retournera la valeur du champ pour le résultat de l'enregistrement actuel. Si aucun enregistrement correspondant n'est trouvé cela retournera false.

```
$this->Post->id = 22;
echo $this->Post->field('name'); // affiche le nom pour la ligne avec l'id 22

echo $this->Post->field('name', array('created <' => date('Y-m-d H:i:s')), 'created DESC');
// affiche le nom de la dernière instance créée
```

Model::read()

`read($fields, $id)`

`read()` est une méthode utilisée pour récupérer les données du model courant (`Model::$data`) - comme lors des mises à jour - mais elle peut aussi être utilisée dans d'autres circonstances, pour récupérer un seul enregistrement depuis la base de données.

`$fields` est utilisé pour passer un seul nom de champ sous forme de chaîne ou un tableau de noms de champs ; si laissé vide, tous les champs seront retournés.

`$id` précise l'ID de l'enregistrement à lire. Par défaut, l'enregistrement actuellement sélectionné, tel que spécifié par `Model::$id`, est utilisé. Passer une valeur différente pour `$id` fera que l'enregistrement correspondant sera sélectionné.

`read()` retourne toujours un tableau (même si seulement un nom de champ unique est requis). Utilisez `field` pour retourner la valeur d'un seul champ.

> **Warning :** Puisque la méthode `read` écrase toute information stockée dans les propriétés `data` and `id` du model, vous devez faire très attention quand vous utilisez cete fonction en général, spécialement en l'utilisant dans les fonctions de callbacks du model comme `beforeValidate` et `beforeSave`. Généralement la fonction `find` est une façon de faire plus robuste et facile à utiliser avec l'API que la méthode `read`.

Conditions de recherche complexes

La plupart des appels de recherche de models impliquent le passage d'un jeu de conditions d'une manière ou d'une autre. Le plus simple est d'utiliser un bout de clause WHERE SQL. Si vous vous avez besoin de plus de contrôle, vous pouvez utiliser des tableaux.

L'utilisation de tableaux est plus claire et simple à lire, et rend également la construction de requêtes très simple. Cette syntaxe sépare également les éléments de votre requête (champs, valeurs, opérateurs etc.) en parties manipulables et discrètes. Cela permet à CakePHP de générer les requêtes les plus efficaces possibles, d'assurer une syntaxe SQL correcte, et d'échapper convenablement chaque partie de la requête. Utiliser une syntaxe en tableau permet aussi à CakePHP de sécuriser vos requêtes contre toute attaque d'injection SQL.

Dans sa forme la plus simple, une requête basée sur un tableau ressemble à ceci :

```
$conditions = array("Post.title" => "This is a post", "Post.author_id" => 1);
// Exemple d'utilisation avec un model:
$this->Post->find('first', array('conditions' => $conditions));
```

La structure ici est assez significative : Tous les posts dont le titre à pour valeur « Ceci est un post » sont cherchés. Nous aurions pu uniquement utiliser « titre » comme nom de champ, mais lorsque l'on construit des requêtes, il vaut mieux toujours spécifier le nom du model. Cela améliore la clarté du code, et évite des collisions futures, dans le cas où vous devriez changer votre schéma.

Qu'en est-il des autres types de correspondances ? Elles sont aussi simples. Disons que nous voulons trouver tous les posts dont le titre n'est pas "Ceci est un post" :

```
array("Post.titre !=" => "Il y a un post")
```

Notez le ' !=' qui précède l'expression. CakePHP peut parser tout opérateur de comparaison valide de SQL, même les expressions de correspondance utilisant LIKE, BETWEEN, ou REGEX, tant que vous laissez un espace entre l'opérateur et la valeur. La seule exception à ceci sont les correspondance du genre IN(...). Admettons que vous vouliez trouver les posts dont le titre appartient à un ensemble de valeur données :

```
array(
    "Post.titre" => array("Premier post", "Deuxième post", "Troisième post")
)
```

Faire un NOT IN(...) correspond à trouver les posts dont le titre n'est pas dans le jeu de données passé :

```
array(
    "NOT" => array("Post.titre" => array("Premier post", "Deuxième post", "Troisième post")
)
```

Ajouter des filtres supplémentaires aux conditions est aussi simple que d'ajouter des paires clé/valeur au tableau :

```
array (
    "Post.titre" => array("Premier post", "Deuxième post", "Troisième post"),
    "Post.created >" => date('Y-m-d', strtotime("-2 weeks"))
)
```

Vous pouvez également créer des recherches qui comparent deux champs de la base de données :

```
array("Post.created = Post.modified")
```

L'exemple ci-dessus retournera les posts où la date de création est égale à la date de modification (par ex les posts qui n'ont jamais été modifiés sont retournés).

Souvenez-vous que si vous vous trouvez dans l'incapacité de formuler une clause WHERE par cette méthode (ex. opérations booléennes), il vous est toujours possible de la spécifier sous forme de chaîne comme ceci :

```
array(
    'Model.champ & 8 = 1',
    // autres conditions habituellement utilisées
)
```

Par défaut, CakePHP fournit les conditions multiples avec l'opérateur booléen AND, ce qui signifie que le bout de code ci-dessus correspondra uniquement aux posts qui ont été créés durant les deux dernières semaines, et qui ont un titre correspondant à ceux donnés. Cependant, nous pouvons simplement trouver les posts qui correspondent à l'une ou l'autre des conditions :

```
array("OR" => array(
    "Post.titre" => array("Premier post", "Deuxième post", "Troisième post"),
    "Post.created >" => date('Y-m-d', strtotime("-2 weeks"))
))
```

Cake accepte toute opération booléenne SQL valide, telles que AND, OR, NOT, XOR, etc., et elles peuvent être en majuscule comme en minuscule, comme vous préférez. Ces conditions sont également infiniment "IMBRIQUABLES". Admettons que vous ayez une relation hasMany/belongsTo entre Posts et Auteurs, ce qui reviendrait à un LEFT JOIN. Admettons aussi que vous vouliez trouver tous les posts qui contiennent un certain mot-clé "magique" ou qui a été créé au cours des deux dernières semaines, mais que vous voulez restreindre votre recherche aux posts écrits par Bob :

```
array(
    "Auteur.nom" => "Bob",
    "OR" => array(
        "Post.titre LIKE" => "%magic%",
        "Post.created >" => date('Y-m-d', strtotime("-2 weeks"))
    )
)
```

Si vous avez besoin de mettre plusieurs conditions sur le même champ, comme quand vous voulez faire une recherche LIKE avec des termses multiples, vous pouvez faire ceci en utilisant des conditions identiques à :

```
array('OR' => array(
    array('Post.titre LIKE' => '%one%'),
    array('Post.titre LIKE' => '%two%')
))
```

Cake peut aussi vérifier les champs null. Dans cet exemple, la requête retournera les enregistrements où le titre du post n'est pas null :

```
array("NOT" => array(
        "Post.titre" => null
    )
)
```

Pour gérer les requêtes BETWEEN, vous pouvez utiliser ceci :

```
array('Post.read_count BETWEEN ? AND ?' => array(1,10))
```

Note : CakePHP quotera les valeurs numériques selon le type du champ dans votre base de données.

Qu'en est-il de GROUP BY ? :

```
array(
    'fields' => array(
        'Produit.type',
        'MIN(Produit.prix) as prix'
    ),
    'group' => 'Produit.type'
)
```

Les données retournées seront dans le format suivant :

```
Array
(
    [0] => Array
    (
        [Produit] => Array
        (
            [type] => Vetement
        )
        [0] => Array
        (
            [prix] => 32
        )
    )
    [1] => Array
    ...
```

Un exemple rapide pour faire une requête DISTINCT. Vous pouvez utiliser d'autres opérateurs, comme MIN(), MAX(), etc..., d'une manière analogue :

```
array(
    'fields' => array('DISTINCT (User.nom) AS nom_de_ma_colonne'),'),
    'order' = >array('User.id DESC' )
)
```

Vous pouvez créer des conditions très complexes, en regroupant des tableaux de conditions multiples :

```
array(
    'OR' => array(
        array('Entreprise.nom' => 'Futurs Gains'),
        array('Entreprise.ville' => 'CA')
    ),
    'AND' => array(
        array(
            'OR' => array(
                array('Entreprise.status' => 'active'),
                'NOT' => array(
                    array('Entreprise.status' => array('inactive', 'suspendue'))
                )
            )
        )
    )
)
```

Qui produira la requête SQL suivante :

```
SELECT `Entreprise`.`id`, `Entreprise`.`nom`,
`Entreprise`.`description`, `Entreprise`.`location`,
`Entreprise`.`created`, `Entreprise`.`status`, `Entreprise`.`taille`

FROM
   `entreprises` AS `Entreprise`
WHERE
```

```
    (('Entreprise'.'nom' = 'Futurs Gains')
    OR
    ('Entreprise'.'nom' = 'Le truc qui marche bien'))
AND
    (('Entreprise'.'status' = 'active')
    OR (NOT ('Entreprise'.'status' IN ('inactive', 'suspendue'))))
```

Sous requêtes

Par exemple, imaginons que nous ayons une table "users" avec "id", "nom" et "statuts". Le statuts peut être "A", "B" ou "C". Et nous voulons récupérer tous les users qui ont un statuts différent de "B" en utilisant une sous requête.

Pour pouvoir effectuer cela, nous allons appeler la source de données du model et lui demander de construire la requête comme si nous appelions une méthode "find", mais elle retournera uniquement la commande SQL. Après cela, nous construisons une expression et l'ajoutons au tableau des conditions :

```
$conditionsSubQuery['"User2"."status"'] = 'B';

$db = $this->Utilisateur->getDataSource();
$subQuery = $db->buildStatement(
    array(
        'fields'     => array('"User2"."id"'),
        'table'      => $db->fullTableName($this->User),
        'alias'      => 'User2',
        'limit'      => null,
        'offset'     => null,
        'joins'      => array(),
        'conditions' => $conditionsSubQuery,
        'order'      => null,
        'group'      => null
    ),
    $this->User
);
$subQuery = ' "User"."id" NOT IN (' . $subQuery . ') ';
$subQueryExpression = $db->expression($subQuery);

$conditions[] = $subQueryExpression;

$this->User->find('all', compact('conditions'));
```

Ceci devrait générer la commande SQL suivante :

```
SELECT
    "User"."id" AS "User__id",
    "User"."name" AS "User__nom",
    "User"."status" AS "User__status"
FROM
    "users" AS "User"
WHERE
    "User"."id" NOT IN (
```

```
        SELECT
            "User2"."id"
        FROM
            "users" AS "User2"
        WHERE
            "User2"."status" = 'B'
    )
```

Aussi, si vous devez passer juste une partie de votre requête en colonne SQL comme ci-dessus, la source de données **expressions** avec la colonne SQL fonctionne pour toute partie de requête find.

Requêtes Préparées

Si vous avez besoin d'encore plus de contrôle sur vos requêtes, vous pouvez utiliser des requêtes préparées. Cela vous permet de parler directement au driver de la base de données et d'envoyer toute requête personnalisée que vous souhaitez :

```
$db = $this->getDataSource();
$db->fetchAll(
    'SELECT * from users where username = ? AND password = ?',
    array('jhon', '12345')
);
$db->fetchAll(
    'SELECT * from users where username = :username AND password = :password',
    array('username' => 'jhon','password' => '12345')
);
```

Sauvegarder vos Données

CakePHP rend la sauvegarde des données d'un model très rapide. Les données prêtes à être sauvegardées doivent être passées à la méthode `save()` du model en utilisant le format basique suivant

```
Array
(
    [NomDuModele] => Array
    (
        [nomduchamp1] => 'valeur'
        [nomduchamp2] => 'valeur'
    )
)
```

La plupart du temps vous n'aurez même pas à vous préoccuper de ce format : le `FormHelper` et les méthodes de recherche de CakePHP réunissent les données sous cette forme. Si vous utilisez un de ces helpers, les données sont également disponibles dans `$this->request->data` pour un usage rapide et pratique.

Voici un exemple simple d'une action de controller qui utilise un model CakePHP pour sauvegarder les données dans une table de la base de données

```
public function edit($id) {
    //Est-ce que des données de formulaires ont été POSTées ?
    if ($this->request->is('post')) {
       //Si les données du formulaire peuvent être validées et sauvegardées ...
       if($this->Recipe->save($this->request->data)) {
           //On définit une message flash en session et on redirige.
           $this->Session->setFlash("Recipe sauvegardée !");
           $this->redirect('/recettes');
       }
    }
    //Si aucune données de formulaire, on récupère la recipe à éditer
    //et on la passe à la vue
    $this->set('recipe', $this->Recipe->findById($id));
}
```

Quand save() est appelée, la donnée qui lui est passée en premier paramètre est validée en utilisant le mécanisme de validation de CakePHP (voir le chapitre *Validation des Donnnées* pour plus d'informations). Si pour une raison quelconque vos données ne se sauvegardent pas, pensez à regarder si des règles de validation ne sont pas insatisfaites. Vous pouvez débugger cette situation en affichant `Model::$validationErrors`:

```
if ($this->Recipe->save($this->request->data)) {
    // Traite le succès.
}
debug($this->Recipe->validationErrors);
```

Il y a quelques autres méthodes du model liées à la sauvegarde que vous trouverez utiles :

`Model::set($one, $two = null)`

`Model::set()` peut être utilisé pour définir un ou plusieurs champs de données du tableau de donnés à l'intérieur d'un Model. C'est utile pour l'utilisation de models avec les fonctionnalités ActiveRecord offertes par le model :

```
$this->Post->read(null, 1);
$this->Post->set('title', 'Nouveau titre pour l\'article');
$this->Post->save();
```

Dans un exemple de l'utilisation de `set()` pour mettre à jour et sauvegarder les champs uniques, dans une approche ActiveRecord. Vous pouvez aussi utiliser `set()` pour assigner de nouvelles valeurs aux champs multiples :

```
$this->Post->read(null, 1);
$this->Post->set(array(
    'title' => 'Nouveau titre',
    'published' => false
));
$this->Post->save();
```

Ce qui est au-dessus met à jour les champs title et published et les sauvegarde dans le base de données.

Model::clear()

Cette méthode peut être utilisée pour réinitialiser l'état du model et effacer toutes les données non sauvegardées et les erreurs de validation. Introduit dans la version 2.4.

Model::save(array $data = null, boolean $validate = true, array $fieldList = array())

La méthode ci-dessus sauvegarde des données formatées sous forme tabulaire. Le second paramètre vous permet de mettre de côté la validation, et le troisième vous permet de fournir une liste des champs du model devant être sauvegardés. Pour une sécurité accrue, vous pouvez limiter les champs sauvegardés à ceux listés dans `$fieldList`.

Note : Si `$fieldList` n'est pas fourni, un utilisateur malicieux peut ajouter des champs supplémentaires dans le formulaire de données (si vous n'utilisez pas `SecurityComponent`), et ainsi changer la valeur de champs qui n'étaient pas prévus à l'origine.

La méthode save a aussi une syntaxe alternative :

```
save(array $data = null, array $params = array())
```

Le tableau `$params` peut avoir n'importe quelle option disponible suivante en clé :
- `validate` Défini à true/false pour activer/désactiver la validation.
- `fieldList` Un tableau de champs que vous souhaitez autoriser pour la sauvegarde.
- `callbacks` Défini à false permet la désactivation des callbacks. En utilisant 'before' ou 'after' activera seulement ces callbacks.
- `counterCache` (depuis 2.4) Booléen pour contrôler la mise à jour des counter caches (si il y en a)

Plus d'informations sur les callbacks du model sont disponibles *ici*.

Astuce : Si vous ne voulez pas le que champ `modified` soit mis à jour pendant la sauvegarde de certaines données, ajoutez `'modified' => false` à votre tableau de `$data`.

Une fois qu'une sauvegarde est terminée, l'ID de l'objet peut être trouvé dans l'attribut `$id` de l'objet Model - quelque chose de spécialement pratique quand on crée de nouveaux objets.

```
$this->Ingredient->save($nouvellesDonnees);
$nouvelIngredientId = $this->Ingredient->id;
```

La création ou la mise à jour est contrôlée par le champ `id` du model. Si `$Model->id` est défini, l'enregistrement avec cette clé primaire est mis à jour. Sinon, un nouvel enregistrement est créé :

```
// Création: id n'est pas défini ou est null
$this->Recipe->create();
$this->Recipe->save($this->request->data);

// Mise à jour: id est défini à une valeur numérique
$this->Recipe->id = 2;
$this->Recipe->save($this->request->data);
```

Astuce : Lors de l'appel à save() dans une boucle, n'oubliez pas d'appeler `create()`.

Si vous voulez mettre à jour une valeur, plutôt qu'en créer une, assurez-vous que vous avez passé le champ de la clé primaire dans le tableau data :

```
$data = array('id' => 10, 'title' => 'Mon Nouveau Titre');
// Cela mettra à jour la Recipe avec un id 10
$this->Recipe->save($data);
```

Model::create(array $data = array())

Cette méthode initialise la classe du model pour sauvegarder de nouvelles informations. Cela ne crée pas réellement un enregistrement dans la base de données mais efface Model : :$id et défini Model : :$data basé sur les champs par défaut dans votre base de données. Si vous n'avez défini aucun champ par défaut dans votre base de données, Model : :$data sera défini comme un tableau vide.

Si le paramètre `$data` (utilisant le format de tableau souligné ci-dessus) est passé, il sera fusionné avec les champs par défaut de la base de données et l'instance du model sera prête à être sauvegardée avec ces données (accessible dans `$this->data`).

Si `false` ou `null` sont passés pour le paramètre `$data`, Model : :data sera défini comme un tableau vide.

Astuce : Si vous voulez insérer une nouvelle ligne au lieu de mettre à jour une ligne existante, vous devriez toujours appeler en premier lieu create(). Cela évite les conflits avec d'éventuels appels à save en amont dans les callbacks ou à tout autre endroit.

Model::saveField(string $fieldName, string $fieldValue, $validate = false)

Utilisée pour sauvegarder la valeur d'un seul champ. Fixez l'ID du model (`$this->ModelName->id = $id`) juste avant d'appeler `saveField()`. Lors de l'utilisation de cette méthode, `$fieldName` ne doit contenir que le nom du champ, pas le nom du model et du champ.

Par exemple, pour mettre à jour le titre d'un article de blog, l'appel depuis un controller à `saveField` ressemblerait à quelque chose comme :

```
$this->Post->saveField('title', 'Un nouveau titre pour un Nouveau Jour');
```

Warning : Vous ne pouvez pas arrêter la mise à jour du champ `modified` avec cette méthode, vous devrez utiliser la méthode save().

La méthode saveField a aussi une syntaxe alternative :

```
saveField(string $fieldName, string $fieldValue, array $params = array())
```

Le tableau `$params` peut avoir en clé, les options disponibles suivantes :

- `validate` Définie à true/false pour activer/désactiver la validation.
- `callbacks` Définie à false pour désactiver les callbacks. Utiliser 'before' ou 'after' activera seulement ces callbacks.
- `counterCache` (depuis 2.4) Booléen pour contrôler la mise à jour des counter caches (si il y en a)

`Model::updateAll(array $fields, array $conditions)`

Met à jour plusieurs enregistrements en un seul appel. Les enregistrements à mettre à jour sont identifiés par le tableau `$conditions`, et les champs devant être mis à jour, ainsi que leurs valeurs, sont identifiés par le tableau `$fields`.

Par exemple, si je voulais approuver tous les boulangers qui sont membres depuis plus d'un an, l'appel à update devrait ressembler à quelque chose du style :

```
$this_year = date('Y-m-d h:i:s', strtotime('-1 year'));

$this->Baker->updateAll(
    array('Baker.approve' => true),
    array('Baker.created <=' => $cette_annee)
);
```

Astuce : Le tableau $fields accepte des expressions SQL. Les valeurs littérales doivent être manuellement quotées en utilisant `Sanitize::escape()`.

Note : Même si le champ modifié existe pour le model qui vient d'être mis à jour, il ne sera pas mis à jour automatiquement par l'ORM. Ajoutez le seulement manuellement au tableau si vous avez besoin de le mettre à jour.

Par exemple, pour fermer tous les tickets qui appartiennent à un certain client :

```
$this->Ticket->updateAll(
    array('Ticket.status' => "'closed'"),
    array('Ticket.client_id' => 453)
);
```

Par défaut, updateAll() joindra automatiquement toute association belongsTo pour les bases de données qui suportent la jointure. Pour éviter cela, délier les associations temporairement.

`Model::saveMany(array $data = null, array $options = array())`

La méthode utilisée pour sauvegarder les lignes multiples du même model en une fois. Les options suivantes peuvent être utilisées :

- `validate` : Défini à false pour désactiver la validation, true pour valider chaque enregistrement avant la sauvegarde, 'first' pour valider *tous* les enregistrements avant qu'un soit sauvegardé (par défaut),
- `atomic` : Si true (par défaut), essaiera de sauvegarder tous les enregistrements en une seule transaction. Devrait être défini à false si la base de données/table ne supporte pas les transactions.

– `fieldList` : Equivalent au paramètre $fieldList dans Model : :save()
– `deep` : (since 2.1) Si défini à true, les données associées sont aussi sauvegardées, regardez aussi saveAssociated.
– `counterCache` (depuis 2.4) Booléen pour contrôler la mise à jour des counter caches (si il y en a)

Pour sauvegarder de multiples enregistrements d'un unique model, $data a besoin d'être un tableau d'enregistrements indexé numériquement comme ceci :

```
$data = array(
    array('title' => 'titre 1'),
    array('title' => 'titre 2'),
)
```

Note : Notez que nous passons les indices numériques de la variable habituelle `$data` contenant le clé Article. Quand vous passez plusieurs enregistrements du même model, les tableaux d'enregistrements doivent être seulement indexés numériquement sans la clé model.

Il est aussi possible d'avoir les données dans le format suivant :

```
$data = array(
    array('Article' => array('title' => 'title 1')),
    array('Article' => array('title' => 'title 2')),
)
```

Pour sauvegarder les données associées avec `$options['deep'] = true` (depuis 2.1), les deux exemples ci-dessus ressembleraient à cela :

```
$data = array(
    array('title' => 'title 1', 'Assoc' => array('field' => 'value')),
    array('title' => 'title 2'),
)
$data = array(
    array('Article' => array('title' => 'title 1'), 'Assoc' => array('field' => 'value')),
    array('Article' => array('title' => 'title 2')),
)
$Model->saveMany($data, array('deep' => true));
```

Gardez à l'esprit que si vous souhaitez mettre à jour un enregistrement au lieu d'en créer un nouveau, vous devez juste ajouter en index la clé primaire à la ligne de donnée :

```
array(
    array('Article' => array('title' => 'New article')), // Ceci crée une nouvelle ligne
    array('Article' => array('id' => 2, 'title' => 'title 2')), // Ceci met à jour une lig
)
```

`Model::saveAssociated(array $data = null, array $options = array())`

Méthode utilisée pour sauvegarder des associations de model en une seule fois. Les options suivantes peuvent être utilisées :

- `validate` : Définie à false pour désactiver la validation, true pour valider chaque enregistrement avant sauvegarde, 'first' pour valider *tous* les enregistrements avant toute sauvegarde (par défaut).
- `atomic` : Si à true (par défaut), va tenter de sauvegarder tous les enregistrements en une seule transaction. Devrait être défini à false si la base de données/table ne supporte pas les transactions.
- `fieldList` : Equivalent au paramètre $fieldList de Model : :save().
- `deep` : (depuis 2.1) Si défini à true, les données pas seulement associées directement vont être sauvegardées, mais aussi les données associées imbriquées plus profondément. Par défaut à false.
- `counterCache` (depuis 2.4) Booléen pour contrôler la mise à jour des counter caches (si il y en a)

Pour sauvegarder un enregistrement et tous ses enregistrements liés avec une association hasOne ou belongsTo, le tableau de données devra ressembler à cela :

```
array(
    'User' => array('username' => 'billy'),
    'Profile' => array('sex' => 'Male', 'occupation' => 'Programmer'),
)
```

Pour sauvegarder un enregistrement et ses enregistrements liés avec une association hasMany, le tableau de données devra ressembler à cela :

```
$data = array(
    'Article' => array('title' => 'My first article'),
    'Comment' => array(
        array('body' => 'Comment 1', 'user_id' => 1),
        array('body' => 'Comment 2', 'user_id' => 12),
        array('body' => 'Comment 3', 'user_id' => 40),
    ),
);
```

Et pour sauvegarder un enregistrement avec ses enregistrements liés par hasMany qui ont plus de deux niveaux d'association de profondeur, le tableau de données devra être comme suit :

```
$data = array(
     'User' => array('email' => 'john-doe@cakephp.org'),
     'Cart' => array(
         array(
             'payment_status_id' => 2,
             'total_cost' => 250,
             'CartItem' => array(
                 array(
                     'cart_product_id' => 3,
                     'quantity' => 1,
                     'cost' => 100,
                 ),
                 array(
                     'cart_product_id' => 5,
                     'quantity' => 1,
                     'cost' => 150,
                 )
             )
         )
     )
 );
```

Note : Si cela réussit, la clé étrangère du model principal va être stockée dans le champ id du model lié, par ex : `$this->RelatedModel->id`.

Warning : Attention quand vous vérifiez les appels saveAssociated avec l'option atomic définie à false. Elle retourne un tableau au lieu d'un boléen.

Modifié dans la version 2.1 : Vous pouvez maintenant aussi sauvegarder les données associées avec la configuration `$options['deep'] = true;`. Pour sauvegarder un enregistrement et ses enregistrements liés avec une association hasMany ainsi que les données associées plus profondément de type Comment belongsTo User, le tableau de données devra ressembler à ceci :

```
$data = array(
    'Article' => array('title' => 'My first article'),
    'Comment' => array(
        array('body' => 'Comment 1', 'user_id' => 1),
        array('body' => 'Sauvegarder aussi un nouveau user', 'User' => array('first' => 'ma
    ),
)
```

Et sauvegarder cette donnée avec :

```
$Article->saveAssociated($data, array('deep' => true));
```

Modifié dans la version 2.1 : `Model::saveAll()` et ses amis supportent maintenant qu'on leur passe *fieldList* pour des models multiples. Exemple d'utilisation de `fieldList` avec de multiples models :

```
$this->SomeModel->saveAll($data, array(
    'fieldList' => array(
        'SomeModel' => array('field_1'),
        'AssociatedModel' => array('field_2', 'field_3')
    )
));
```

La fieldList sera un tableau d'alias de model en clé et de tableaux avec les champs en valeur. Les noms de model ne sont pas imbriqués comme dans les données à sauvegarder.

`Model::saveAll(array $data = null, array $options = array())`

La fonction `saveAll` est juste un wrapper autour des méthodes `saveMany` et `saveAssociated`. Elle va inspecter les données et déterminer quel type de sauvegarde elle devra effectuer. Si les données sont bien formatées en un tableau indicé numériquement, `saveMany` sera appelée, sinon `saveAssociated` sera utilisée.

Cette fonction reçoit les mêmes options que les deux précédentes, et est généralement une fonction rétro-compatible. Il est recommandé d'utiliser soit `saveMany` soit `saveAssociated` selon le cas.

Sauvegarder les Données de Models Liés (hasOne, hasMany, belongsTo)

Quand vous travaillez avec des models associés, il est important de réaliser que la sauvegarde de données de model devrait toujours être faite avec le model CakePHP correspondant. Si vous sauvegardez un nouveau Post et ses Comments associés, alors vous devriez utiliser les deux models Post et Comment pendant l'opération de sauvegarde.

Si aucun des enregistrements du model associé n'existe pour l'instant dans le système (par exemple, vous voulez sauvegarder un nouveau User et ses enregitrements du Profile lié en même temps), vous aurez besoin de sauvegarder d'abord le model principal, ou le model parent.

Pour avoir une bonne idée de la façon de faire, imaginons que nous ayons une action dans notre UsersController qui gère la sauvegarde d'un nouveau User et son Profile lié. L'action montré en exemple ci-dessous supposera que vous avez POSTé assez de données (en utilisant FormHelper) pour créer un User unique et un Profile unique :

```
public function add() {
    if (!empty($this->request->data)) {
        // Nous pouvons sauvegarder les données de l'User:
        // it should be in $this->request->data['User']

        $user = $this->User->save($this->request->data);

        // Si l\'user a été sauvegardé, maintenant nous ajoutons cette information aux don
        // et sauvegardons le Profile.

        if (!empty($user)) {
            // L'ID de l\'user nouvellement crée a été défini
            // dans $this->User->id.
            $this->request->data['Profile']['user_id'] = $this->User->id;

            // Parce que notre User hasOne Profile, nous pouvons accéder
            // au model Profile à travers le model User:
            $this->User->Profile->save($this->request->data);
        }
    }
}
```

Comme règle, quand vous travaillez avec des associations hasOne, hasMany, et belongsTo, Tout est question de clé. L'idée de base est de récupérer la clé d'un autre model et de la placer dans le champ clé étrangère sur l'autre. Parfois, cela pourra gêner l'utilisation de l'attribut `$id` de la classe model après un `save()`, mais d'autres fois, cela impliquera juste la collecte de l'ID provenant d'un champ caché d'un formulaire qui vient d'être POSTé d'une action d'un controller.

Pour compléter l'approche fondamentale utilisée ci-dessus, CakePHP offre également une méthode très pratique `saveAssociated()`, qui vous permet de valider et de sauvegarder de multiples models en une fois. De plus, `saveAssociated()` fournit un support transactionnel pour s'assurer de l'intégrité des données dans votre base de données (par ex : si un model échoue dans la sauvegarde, les autres models ne seront également pas sauvegardés).

Note : Pour que les transactions fonctionnent correctement dans MySQL, vos tables doivent utiliser le moteur InnoDB. Souvenez-vous que les tables MyISAM ne supportent pas les transactions.

Voyons comment nous pouvons utiliser `saveAssociated()` pour sauvegarder les models Company et Account en même temps.

Tout d'abord, vous avez besoin de construire votre formulaire pour les deux models Company and Account (nous supposerons que Company hasMany Account) :

```
echo $this->Form->create('Company', array('action' => 'add'));
echo $this->Form->input('Company.name', array('label' => 'Company name'));
echo $this->Form->input('Company.description');
echo $this->Form->input('Company.location');

echo $this->Form->input('Account.0.name', array('label' => 'Account name'));
echo $this->Form->input('Account.0.username');
echo $this->Form->input('Account.0.email');

echo $this->Form->end('Add');
```

Regardez comment nous avons nommé les champs de formulaire pour le model Account. Si Company est notre model principal, `saveAssociated()` va s'attendre à ce que les données du model lié (Account) arrivent dans un format spécifique. Et avoir `Account.0.fieldName` est exactement ce dont nous avons besoin.

Note : Le champ ci-dessus est nécessaire pour une association hasMany. Si l'association entre les models est hasOne, vous devrez utiliser la notation ModelName.fieldName pour le model associé.

Maintenant, dans notre CompaniesController nous pouvons créer une action `add()` :

```
public function add() {
    if (!empty($this->request->data)) {
        // Utilisez ce qui suit pour éviter les erreurs de validation:
        unset($this->Company->Account->validate['company_id']);
        $this->Company->saveAssociated($this->request->data);
    }
}
```

C'est tout pour le moment. Maintenant nos models Company et Account seront validés et sauvegardé en même temps. Par défaut `saveAssociated` validera toutes les valeurs passées et ensuite essaiera d'effectuer une sauvegarde pour chacun.

Sauvegarder hasMany through data

Regardons comment les données stockées dans une table jointe pour deux models sont sauvegardées. Comme montré dans la section *hasMany through (Le Model Join)*, la table jointe est associée pour chaque model en utilisant un type de relation *hasMany*. Notre exemple est une problématique lancée par la Tête de l'Ecole Cake qui nous demande d'écrire une application qui lui permette de connecter la présence d'un étudiant à un cours avec les journées assistées et de validées. Jettez un oeil au code suivant. :

```
// Controller/CourseMembershipController.php
class CourseMembershipsController extends AppController {
    public $uses = array('CourseMembership');

    public function index() {
        $this->set('courseMembershipsList', $this->CourseMembership->find('all'));
    }

    public function add() {
        if ($this->request->is('post')) {
            if ($this->CourseMembership->saveAssociated($this->request->data)) {
                $this->redirect(array('action' => 'index'));
            }
        }
    }
}

// View/CourseMemberships/add.ctp

<?php echo $this->Form->create('CourseMembership'); ?>
    <?php echo $this->Form->input('Student.first_name'); ?>
    <?php echo $this->Form->input('Student.last_name'); ?>
    <?php echo $this->Form->input('Course.name'); ?>
    <?php echo $this->Form->input('CourseMembership.days_attended'); ?>
    <?php echo $this->Form->input('CourseMembership.grade'); ?>
    <button type="submit">Save</button>
<?php echo  $this->Form->end(); ?>
```

Le tableau de données ressemblera à ceci quand il sera soumis. :

```
Array
(
    [Student] => Array
    (
        [first_name] => Joe
        [last_name] => Bloggs
    )

    [Course] => Array
    (
        [name] => Cake
    )

    [CourseMembership] => Array
    (
        [days_attended] => 5
        [grade] => A
    )

)
```

Cake va heureusement être capable de sauvegarder le lot ensemble et d'assigner les clés étrangères de Student et de Course dans CourseMembership avec un appel *saveAssociated* avec cette structure de données.

Si nous lançons l'action index de notre CourseMembershipsController, la structure de données reçue maintenant par un find('all') est :

```
Array
(
    [0] => Array
    (
        [CourseMembership] => Array
        (
            [id] => 1
            [student_id] => 1
            [course_id] => 1
            [days_attended] => 5
            [grade] => A
        )

        [Student] => Array
        (
            [id] => 1
            [first_name] => Joe
            [last_name] => Bloggs
        )

        [Course] => Array
        (
            [id] => 1
            [name] => Cake
        )
    )
)
```

Il y a bien sûr beaucoup de façons de travailler avec un model joint. La version ci-dessus suppose que vous voulez sauvegarder tout en une fois. Il y aura des cas où vous voudrez créer les Student et Course indépendamment et associer les deux ensemble avec CourseMemberShip plus tard. Donc, vous aurez peut-être un formulaire qui permet la sélection de students et de courses existants à partir d'une liste de choix ou d'une entrée d'un ID et ensuite les deux meta-champs pour CourseMembership, par ex. :

```
// View/CourseMemberships/add.ctp

<?php echo $form->create('CourseMembership'); ?>
    <?php echo $this->Form->input('Student.id', array('type' => 'text', 'label' => 'Student
    <?php echo $this->Form->input('Course.id', array('type' => 'text', 'label' => 'Course I
    <?php echo $this->Form->input('CourseMembership.days_attended'); ?>
    <?php echo $this->Form->input('CourseMembership.grade'); ?>
    <button type="submit">Save</button>
<?php echo $this->Form->end(); ?>
```

Et le POST résultant :

```
Array
(
    [Student] => Array
    (
```

```
        [id] => 1
    )

    [Course] => Array
    (
        [id] => 1
    )

    [CourseMembership] => Array
    (
        [days_attended] => 10
        [grade] => 5
    )
)
```

Encore une fois, Cake est bon pour nous et envoie les id de Student et de Course dans CourseMembership avec *saveAssociated*.

Sauvegarder les Données de Model Lié (HABTM=HasAndBelongsToMany)

Sauvegarder les models qui sont associés avec hasOne, belongsTo, et hasMany est assez simple : vous venez de remplir le champ de la clé étrangère avec l'ID du model associé. Une fois que c'est fait, vous appelez juste la méthode `save()` sur un model, et tout se relie correctement. Un exemple du format requis pour le tableau de données passé à `save()` pour le model Tag model est montré ci-dessous :

```
Array
(
    [Recipe] => Array
        (
            [id] => 42
        )
    [Tag] => Array
        (
            [name] => Italian
        )
)
```

Vous pouvez aussi utiliser ce format pour sauvegarder plusieurs enregistrements et leurs associations HABTM avec `saveAll()`, en utilisant un tableau comme celui qui suit :

```
Array
(
    [0] => Array
        (
            [Recipe] => Array
                (
                    [id] => 42
                )
            [Tag] => Array
                (
                    [name] => Italian
```

```
                )
        )
    [1] => Array
        (
            [Recipe] => Array
                (
                    [id] => 42
                )
            [Tag] => Array
                (
                    [name] => Pasta
                )
        )
    [2] => Array
        (
            [Recipe] => Array
                (
                    [id] => 51
                )
            [Tag] => Array
                (
                    [name] => Mexican
                )
        )
    [3] => Array
        (
            [Recipe] => Array
                (
                    [id] => 17
                )
            [Tag] => Array
                (
                    [name] => American (new)
                )
        )
)
```

Passer le tableau ci-dessus à `saveAll()` va créer les tags contenus, chacun associé avec leur recettes respectives.

Pour exemple, nous allons construire un formulaire qui crée un nouveau tag et générerons le tableau de données approprié pour l'associer à la volée avec certaines recettes.

Le formulaire le plus simple ressemblerait à ceci (nous supposerons que `$recipe_id` est déjà défini à une valeur) :

```
<?php echo $this->Form->create('Tag');?>
    <?php echo $this->Form->input(
        'Recipe.id',
        array('type' => 'hidden', 'value' => $recipe_id)); ?>
    <?php echo $this->Form->input('Tag.name'); ?>
<?php echo $this->Form->end('Add Tag'); ?>
```

Dans cet exemple, vous pouvez voir le champ caché `Recipe.id` dont la valeur est définie selon l'ID de la recette que nous voulons lier au tag.

Quand la méthode `save()` est appelée dans le controller, elle va automatiquement sauvegarder les données HABTM dans la base de données :

```
public function add() {
    // Sauvegarder l'association
    if ($this->Tag->save($this->request->data)) {
        // faire quelque chose en cas de succès
    }
}
```

Avec le code précédent, notre Tag nouveau est crée et associé avec un Recipe, dont l'ID a été défini dans `$this->request->data['Recipe']['id']`.

Les autres façons que nous pouvons avoir envie pour présenter nos données associées peuvent inclure une liste de select drop down. Les données peuvent être envoyées d'un model en utilisant la méthode `find('list')` et assignées à une variable de vue du nom du model. Une entrée avec le même nom va automatiquement envoyer dans ces données dans un `<select>` :

```
// dans le controller:
$this->set('tags', $this->Recipe->Tag->find('list'));

// dans la vue:
$form->input('tags');
```

Un scénario plus probable avec une relation HABTM incluerait un `<select>` défini pour permettre des sélections multiples. Par exemple, un Recipe peut avoir plusieurs Tags lui étant assignés. Dans ce cas, les données sort du model de la même façon, mais l'entrée du formulaire est déclarée légèrement différemment. Le nom du Tag est défini en utilisant la convention `ModelName` :

```
// dans le controller:
$this->set('tags', $this->Recipe->Tag->find('list'));

// dans la vue:
$this->Form->input('Tag');
```

En utilisant le code précédent, un select drop down est crée, permettant aux multiples choix d'être automatiquement sauvegarder au Recipe existant en étant ajouté à la base de données.

Self HABTM Normalement HABTM est utilisé pour lier 2 models ensemble mais il peut aussi être utilisé avec seulement 1 model, mais il nécéssite une attention plus grande encore.

La clé est dans la configuration du model `className`. En ajoutant simplement une relation `Project` HABTM `Project` entraine des problèmes lors des enregistrements de données. En configurant le `className` au nom de models et en utilisant l'alias en clé, nous évitons ces problèmes. :

```
class Project extends AppModel {
    public $hasAndBelongsToMany = array(
        'RelatedProject' => array(
            'className'              => 'Project',
```

```
            'foreignKey'              => 'projects_a_id',
            'associationForeignKey'   => 'projects_b_id',
        ),
    );
}
```

Créer des éléments de form et sauvegarder les données fonctionne de la même façon qu'avant mais vous utilisez l'alias à la place. Ceci :

```
$this->set('projects', $this->Project->find('list'));
$this->Form->input('Project');
```

Devient ceci :

```
$this->set('relatedProjects', $this->Project->find('list'));
$this->Form->input('RelatedProject');
```

Que faire quand HABTM devient compliqué ? Par défaut, quand vous sauvegardez une relation HasAndBelongsToMany, Cake supprime toutes les lignes de la table jointe avant d'en sauvegarder de nouvelles. Par exemple, si vous avez un Club qui a 10 Children (Enfant) associés. Vous mettez ensuite à jour le Club avec 2 Children. Le Club aura seulement 2 Children, et pas 12.

Notez aussi que si vous voulez ajouter plus de champs à joindre (quand il a été crée ou les meta informations), c'est possible avec les tables jointes HABTM, mais il est important de comprendre que vous avez une option facile.

HasAndBelongsToMany entre deux models est en réalité un raccourci pour trois models associés à travers les deux associations hasMany et belongsTo.

Etudiez cet exemple :

```
Child hasAndBelongsToMany Club
```

Une autre façon de regarder cela est d'ajouter un model Membership :

```
Child hasMany Membership
Membership belongsTo Child, Club
Club hasMany Membership.
```

Ces deux exemples sont presque les mêmes. Ils utilisent le même montant de champs nommés dans la base de données et le même montant de models. Les différences importantes sont que le model "join" est nommé différemment et que son comportement est plus prévisible.

Astuce : Quand votre table jointe contient des champs supplémentaires en plus des deux clés étrangères, vous pouvez éviter de perdre les valeurs des champs supplémentaires en définissant la clé `'unique'` du tableau à `'keepExisting'`. Vous pouvez penser le penser comme quelque chose de similaire à 'unique' => true, mais sans perdre les données des champs supplémentaires pendant l'opération de sauvegarde. Regardez : *les tablaux des associations HABTM*.

Cependant, dans la plupart des cas, il est plus facile de faire un model pour la table jointe et de configurer les associations hasMany, belongsTo comme montré dans l'exemple ci-dessus au lieu d'utiliser une association HABTM.

Datatables

Tandis que CakePHP peut avoir des sources de données qui ne sont pas des driven de base de données, la plupart du temps, elles le sont. CakePHP est pensé pour être agnostique et va fonctionner avec MySQL, MSSQL, PostgreSQL et autres. Vous pouvez créer vos tables de base de données comme vous l'auriez fait normalement. Quand vous créez vos classes Model, elles seront automatiquement liées aux tables que vous avez créees. Les noms de table sont par convention en minuscules et au pluriel avec tous les mots de la table séparés par des underscores. Par exemple, un nom de mode Ingredient s'attendra à un nom de table ingredients. Un nom de Model de EventRegistration s'attendra à un nom de table de event_registrations. CakePHP va inspecter vos tables pour déterminer le type de données de chaque champ et utiliser cette information pour automatiser plusieurs fonctionnalités comme l'affichage des champs de formulaires dans la vue. Les noms de champ sont par convention en minuscules et séparés par des underscores.

Utiliser created et modified

En définissant un champ `created` ou `modified` dans votre table de base de données en type datetime (par défaut à null), CakePHP va reconnaître ces champs et les remplir automatiquement dès qu'un enregistrement est crée ou sauvegardé dans la base de données (à moins que les données déjà sauvegardées contiennent une valeur pour ces champs).

Les champs `created` et `modified` vont être définis à la date et heure courante quand l'enregistrement est ajouté pour la première fois. Le champ modifié sera mis à jour avec la date et l'heure courante dès que l'enregistrement sera sauvegardé.

Si vous avez `created` ou `modified` des données dans votre $this->data (par ex à partir d'un Model : :read ou d'un Model : :set) avant un Model : :save(), alors les valeurs seront prises à partir de $this->data et ne seront pas mises à jour automagiquement. Si vous ne souhaitez pas cela, vous pouvez utiliser `unset($this->data['Model']['modified'])`, etc... Alternativement vous pouvez surcharger Model : :save() pour toujours le faire pour vous :

```
class AppModel extends Model {

    public function save($data = null, $validate = true, $fieldList = array()) {
        // Nettoie la valeur du champ modified avant chaque sauvegarde
        $this->set($data);
        if (isset($this->data[$this->alias]['modified'])) {
            unset($this->data[$this->alias]['modified']);
        }
        return parent::save($this->data, $validate, $fieldList);
    }

}
```

Supprimer des Données

La classe Model de CakePHP offre de nombreuses façons de supprimer des enregistrements de votre base de données.

delete

`delete(int $id = null, boolean $cascade = true);`

Supprime l'enregistrement identifié par $id. Par défaut, supprime également les enregistrements dépendants de l'enregistrement mentionné comme devant être supprimé.

Par exemple, lors de la suppression d'un enregistrement User lié à plusieurs enregistrements Recipe (User 'hasMany' ou 'hasAndBelongsToMany' Recipes) :
- si $cascade est fixé à true, les entrées Recipe liées sont aussi supprimées si les valeurs "dependant" des models sont à true.
- si $cascade est fixé à false, les entrées Recipe resteront après que l'User ait été supprimé.

Si votre base de données permet les clés étrangères et les suppressions en cascade, il est souvent plus efficace de les utiliser plutôt que le cascade de CakePHP. Le seul bénéfice pour l'utilisation de la fonctionnalité de cascade de `Model::delete()` est qu'elle vous permet d'influencer les callbacks des behaviors et des Models :

```
$this->Comment->delete($this->request->data('Comment.id'));
```

Vous pouvez brancher une logique personnalisée dans le processus de suppression à l'aide des callbacks `beforeDelete` et `afterDelete` présents dans les deux Models et Bahaviors. Allez voir *Méthodes Callback* pour plus d'informations.

deleteAll

`deleteAll(mixed $conditions, $cascade = true, $callbacks = false)`

`deleteAll()` est identique à `delete()`, sauf que `deleteAll()` supprimera tous les enregistrements qui matchent les conditions fournies. Le tableau `$conditions` doit être fourni en tant que fragment ou tableau SQL.
- **conditions** Conditions pour matcher.
- **cascade** Booléen, Mis à true pour supprimer les enregistrements qui dépendent de cet enregistrement.
- **callbacks** Booléen, Lance les callbacks

Retourne un booléen True en cas de succès, false en cas d'échec.

Exemple :

```
// Suppression avec un tableau de conditions similaires à find()
$this->Comment->deleteAll(array('Comment.spam' => true), false);
```

Si vous supprimez avec soit callbacks et/ou cascade, les lignes seront trouvées et ensuite supprimées. Cela impliquera souvent plus de requêtes faîtes.

Note : deleteAll() retournera true même si aucun enregistrement n'est supprimé, puisque les conditions pour la requête de suppression est un succès et qu'aucun enregitrement correspondant ne reste.

Validation des Donnnées

La validation des données est une partie importante de toute application, puisqu'elle permet de s'assurer que les données d'un model respectent les règles métiers de l'application. Par exemple, vous aimeriez vérifier que les mots de passe sont longs d'au moins huit caractères ou bien vous assurer que les noms d'users sont uniques. La définition des règles de validation facilite grandement la gestion des formulaires.

Il y a de nombreux aspects différents dans le processus de validation. Ce que nous aborderons dans cette section c'est le côté model des choses. En résumé : ce qui se produit lorsque vous appelez la méthode save() de votre model. Pour obtenir plus d'informations sur la manière d'afficher les erreurs de validation, regardez la section traitant des helpers *FormHelper*.

La première étape pour la validation de données est de créer les règles dans le Model. Pour ce faire, utilisez le tableau Model : :validate dans la définition du model, par exemple :

```
class User extends AppModel {
    public $validate = array();
}
```

Dans l'exemple ci-dessus, le tableau `$validate` est ajouté au model User, mais ce tableau ne contient pas de règles de validation. En supposant que la table "users" ait les champs "login", "password", "email" et "date_de_naissance", l'exemple ci-dessous montre quelques règles simples de validation qui s'appliquent à ces champs :

```
class User extends AppModel {
    public $validate = array(
        'login' => 'alphaNumeric',
        'email' => 'email',
        'date_de_naissance'  => 'date'
    );
}
```

Ce dernier exemple montre comment des règles de validation peuvent être ajoutées aux champs d'un model. Pour le champ 'login', seules les lettres et les chiffres sont autorisés, l'email doit être valide et la date de naissance doit être une date valide. La définition de règles de validation active l'affichage "automagique" de messages d'erreurs dans les formulaires par CakePHP, si les données saisies ne respectent pas les règles définies.

CakePHP a de nombreuses règles et leur utilisation peut être très simple. Certaines de ces règles intégrées vous permettent de vérifier le format des adresses emails, des URLs, des numéros de carte de crédit, etc. - mais nous couvrirons cela en détail plus loin.

Voici un autre exemple de validation plus complexe qui tire profit de quelques-unes de ces règles pré-définies :

```
class User extends AppModel {
    public $validate = array(
        'login' => array(
            'alphaNumeric' => array(
                'rule'     => 'alphaNumeric',
                'required' => true,
                'message'  => 'Chiffres et lettres uniquement !'
            ),
            'between' => array(
                'rule'    => array('between', 5, 15),
                'message' => 'Entre 5 et 15 caractères'
            )
        ),
        'password' => array(
            'rule'    => array('minLength', '8'),
            'message' => '8 caractères minimum'
        ),
        'email' => 'email',
        'date_de_naissance' => array(
            'rule'       => 'date',
            'message'    => 'Entrez une date valide',
            'allowEmpty' => true
        )
    );
}
```

Deux règles de validation sont définies pour le login : il doit contenir des lettres et des chiffres uniquement et sa longueur doit être comprise entre 5 et 15. Le mot de passe doit avoir au minimum 8 caractères. L'email doit avoir un format correct et la date de naissance être une date valide. Vous pouvez voir dans cet exemple comment personnaliser les messages que CakePHP affichera en cas de non respect de ces règles.

Comme le montre l'exemple ci-dessus, un seul champ peut avoir plusieurs règles de validation. Si les règles pré-définies ne correspondent pas à vos critères, vous pouvez toujours ajouter vos propres règles de validation, selon vos besoins.

Maintenant que nous avons vu, en gros, comment la validation fonctionne, voyons comme ces règles sont définies dans le model. Il y a trois manières différentes pour définir les règles de validation : tableaux simples, une règle par champ et plusieurs règles par champ.

Règles simples

Comme le suggère le nom, c'est la manière la plus simple de définir une règle de validation. La syntaxe générale pour définir des règles de cette manière est :

```
public $validate = array('nomChamp' => 'nomRegle');
```

Où 'nomChamp' est le nom du champ pour lequel la règle est définie, et 'nomRegle' est un nom prédéfini, comme 'alphaNumeric', 'email' ou 'isUnique'.

Par exemple, pour s'assurer que l'user fourni une adresse email correcte, vous pouvez utiliser cette règle :

```
public $validate = array('user_email' => 'email');
```

Une règle par champ

Cette technique de définition permet un meilleur contrôle sur le fonctionnement des règles de validation. Mais avant d'aborder ce point, regardons le schéma d'utilisation général pour ajouter une règle à un seul champ :

```
public $validate = array(
    'champ1' => array(
        'rule'        => 'nomRegle', // ou bien : array('nomRegle', 'parametre1', 'parametre
        'required'    => true,
        'allowEmpty'  => false,
        'on'          => 'create', // ou bien: 'update'
        'message'     => 'Votre message d\'erreur'
    )
);
```

La clé 'rule' est obligatoire. Si vous définissez uniquement 'required' => true, la validation du formulaire ne fonctionnera pas correctement. C'est à cause du fait que 'required' n'est pas à proprement parlé une règle.

Comme vous pouvez le voir ici, chaque champ (un seul est présenté ci-dessus) est associé à un tableau contenant cinq clés : 'rule', 'required', 'allowEmpty', 'on' et 'message'. Toutes les clés sont optionnelles sauf 'rule'. Regardons en détail ces clés.

La clé 'rule'

La clé 'rule' définit la méthode de validation et attend soit une valeur simple, soit un tableau. La règle spécifiée peut-être le nom d'une méthode dans votre model, une méthode de la classe globale Validation ou une expression régulière. Pour une liste complète des règles pré-définies, allez voir *Règles de validation incluses*.

Si la règle ne nécessite pas de paramètre, 'rule' peut-être une simple valeur, comme :

```
public $validate = array(
    'login' => array(
        'rule' => 'alphaNumeric'
    )
);
```

Si la règle nécessite quelques paramètres (tels que un maximum, un minimum ou une plage de valeurs), 'rule' doit être un tableau :

```
public $validate = array(
    'password' => array(
        'rule' => array('minLength', 8)
    )
);
```

Souvenez-vous, la clé 'rule' est obligatoire pour les définitions de règles sous forme de tableau.

La clé 'required'

Cette clé doit être définie par une valeur booléenne, ou `create` ou `update`. Si 'required' est `true` alors le champ doit être présent dans le tableau de données. Tandis que mettre le champ à `create` ou `update` rendra le champ nécessaire seulement lors des opérations de création ou de mise à jour. Par exemple, si la règle de validation a été définie comme suit :

```
public $validate = array(
    'login' => array(
        'rule'     => 'alphaNumeric',
        'required' => true
    )
);
```

Les données envoyées à la méthode save() du model doivent contenir des données pour le champ 'login'. Dans le cas contraire, la validation échouera. La valeur par défaut de cette clé est le booléen 'false'.

`required => true` ne signifie pas la même chose que la règle de validation `notEmpty()`. `required => true` indique que la *clé* du tableau doit être présente - cela ne veut pas dire qu'elle doit avoir une valeur. Par conséquent, la validation échouera si le champ n'est pas présent dans le jeu de données, mais pourra réussir (en fonction de la règle) si la valeur soumise est vide ('). Modifié dans la version 2.1 : Le support pour `create` et `update` a été ajouté.

La Clé 'allowEmpty'

Si définie à `false`, la valeur du champ doit être **non vide**, ceci étant déterminé par `!empty($value) || is_numeric($value)`. La vérification numérique est là pour que CakePHP fasse ce qu'il faut quand `$valeur` vaut zéro.

La différence entre `required` et `allowEmpty` peut être confuse. `'required' => true` signifie que vous ne pouvez pas sauvegarder le model, si la *clé* pour ce champ n'est pas présente dans `$this->data` (la vérification est réalisé avec isset) ; tandis que `'allowEmpty' => false` s'assure que la *valeur* du champ courant est "non vide", comme décrit ci-dessus.

La clé 'on'

La clé 'on' peut prendre l'une des valeurs suivantes : 'update' ou 'create'. Ceci fournit un mécanisme qui permet à une règle donnée d'être appliquée pendant la création ou la mise à jour d'un enregistrement.

Si une règle est définie à 'on' => 'create', elle sera seulement appliquée lors de la création d'un nouvel enregistrement. Autrement, si elle est définie à 'on' => 'update', elle s'appliquera uniquement lors de la mise à jour de l'enregistrement.

La valeur par défaut pour 'on' est 'null'. Quand 'on' est null, la règle s'applique à la fois pendant la création et la mise à jour.

La clé 'message'

La clé 'message' vous permet de définir un message d'erreur de validation personnalisé pour la règle :

```
public $validate = array(
    'password' => array(
        'rule'    => array('minLength', 8),
        'message' => 'Le mot de passe doit comporter au moins 8 caractères'
    )
);
```

Note : Quelque soit la règle, l'échec de validation sans un message défini par défaut sera "This field cannot be left blank." (Ce champ ne peut être laissé vide)

Plusieurs règles par champ

La technique que nous venons de voir nous donne plus de flexibilité que l'assignation simple de règles, mais il y a une étape supplémentaire que nous pouvons mettre en œuvre, pour avoir un contrôle encore plus fin sur la validation des données. La prochaine technique que nous allons voir nous permet d'affecter plusieurs règles de validation par champ de model.

Si vous souhaitiez affecter plusieurs règles de validation à un seul champ, voici basiquement comment il faudrait faire :

```
public $validate = array(
    'nomChamp' => array(
        'nomRegle' => array(
            'rule' => 'nomRegle',
            // clés supplémentaires comme 'on', 'required', etc. à mettre ici
        ),
        'nomRegle2' => array(
            'rule' => 'nomRegle2',
            // clés supplémentaires comme 'on', 'required', etc. à mettre ici
        )
    )
);
```

Comme vous pouvez le voir, cela ressemble beaucoup à ce que nous avons vu dans la section précédente. Ici pour chaque champ, nous avons uniquement un tableau de paramètres de validation. Dans ce cas, chaque 'nomChamp' est un tableau de règles indexé. Chaque 'nomRegle' contient un tableau indépendant de paramètres de validation.

Ce sera plus explicite avec un exemple pratique :

```
public $validate = array(
    'login' => array(
        'regleLogin-1' => array(
            'rule'    => 'alphaNumeric',
            'message' => 'Lettres et chiffres uniquement',
            'last'    => true
```

```
        ),
        'regleLogin-2' => array(
            'rule'    => array('minLength', 8),
            'message' => 'Taille minimum de 8 caractères'
        )
    )
);
```

L'exemple ci-dessus définit deux règles pour le champ 'login' : 'regleLogin-1' et 'regleLogin-2'. Comme vous pouvez le voir, chaque règle est identifiée avec un nom arbitraire.

Quand vous utilisez des règles multiples par champ, les clés 'required' et 'allowEmpty' doivent être utilisées seulement une fois dans la première règle.

La clé 'last'

Dans le cas de règles multiples par champ, si une des règles échoue, le message d'erreur pour cette règle va par défaut être retourné et les règles suivantes pour ce champ ne seront pas testées. Si vous voulez que la validation continue bien qu'une règle ait échouée, définissez la clé `last` à `false` pour cette règle.

Dans l'exemple suivant, même si "rule1" échoue "rule2" va être testée et les messages d'erreur pour les deux règles ayant échoués seront retournées si "rule2" échoue aussi :

```
public $validate = array(
    'login' => array(
        'rule1' => array(
            'rule'    => 'alphaNumeric',
            'message' => 'Only alphabets and numbers allowed',
            'last'    => false
        ),
        'rule2' => array(
            'rule'    => array('minLength', 8),
            'message' => 'Minimum length of 8 characters'
        )
    )
);
```

Quand vous spécifiez des règles de validation dans ce tableau de formulaire, il est aussi possible d'éviter de fournir la clé `message`. Regardez cette exemple :

```
public $validate = array(
    'login' => array(
        'Only alphabets and numbers allowed' => array(
            'rule'    => 'alphaNumeric',
        ),
    )
);
```

Si les règles de `alphaNumeric` échouent, la clé du tableau pour cette règle 'Only alphabets and numbers allowed' sera retourné en message d'erreur si la clé `message` n'est pas définie.

Règles personnalisées de validation des données

Si ce qui précède ne vous convient pas, vous pouvez toujours créer vos propres règles de validation. Il y a deux moyens de réaliser cela : en définissant des expressions régulières ou en créant des méthodes de validation personnalisées.

Validation avec Expression Régulière personnalisée

Si la technique de validation dont vous avez besoin peut être complétée par l'utilisation d'une expression régulière, vous pouvez définir une expression personnalisée comme une règle de validation de champ :

```
public $validate = array(
    'login' => array(
        'rule'    => '/^[a-z0-9]{3,}$/i',
        'message' => 'Seulement des lettres et des entiers, minimum 3 caractères'
    )
);
```

L'exemple ci-dessus vérifie que le login contient seulement des lettres et des entiers et qu'il a au minimum trois caractères.

L'expression régulière dans `rule` doit être délimitée par des slashes (/). Le 'i' final optionnel après le dernier slash signifie que l'expression régulière est *i*nsensible à la casse.

Ajouter vos propres méthodes de validation

Parfois, la vérification des données par un motif d'expression régulière ne suffit pas. Par exemple, si vous voulez vous assurer qu'un coupon de réduction (code promo) n'est pas utilisé plus de 25 fois, vous devez ajouter votre propre méthode de validation, comme indiqué ci-dessous :

```
class User extends AppModel {
    public $validate = array(
        'code_promo' => array(
            'rule'    => array('limiteUtilisations', 25),
            'message' => 'Ce code promo a dépassé son nombre maximal d\'utilisation.'
        )
    );

    public function limiteUtilisations($check, $limit) {
        // $check aura comme valeur : array('code_promo' => 'une valeur')
        // $limit aura comme valeur : 25
         $compteur_code_actuel = $this->find( 'count', array(
            'conditions' => $check,
            'recursive' => -1
        ));
        return $compteur_code_actuel < $limit;
    }
}
```

Le champ en cours de validation est passé à la fonction comme premier paramètre, sous la forme d'un tableau associatif avec le nom du champ comme clé et les données postées comme valeur.

Si vous voulez passer des paramètres supplémentaires à votre fonction de validation, ajoutez des éléments dans le tableau 'rule' et manipulez-les comme des paramètres supplémentaires (après le paramètre principal `$check`) dans votre fonction.

Votre fonction de validation peut être dans le model (comme dans l'exemple) ou dans un behavior (comportement) que votre model implémente. Ceci inclut les méthodes mappées.

Les méthodes des models/behaviors sont vérifiées en premier, avant de chercher pour une méthode dans la classe `Validation`. Cela veut dire que vous pouvez surcharger les méthodes de validation existantes (telle que `alphaNumeric()`) au niveau de l'application (en ajoutant la méthode dans `AppModel`) ou au niveau du model.

Quand vous écrivez une règle de validation qui peut être utilisée par plusieurs champs, prenez soin d'extraire la valeur du champ du tableau $check. Le tableau $check est passé avec le nom du champ comme clé et la valeur du champ comme valeur. Le champ complet qui doit être validé est stocké dans une variable de $this->data :

```
class Post extends AppModel {
    public $validate = array(
        'slug' => array(
            'rule'    => 'alphaNumericDashUnderscore',
            'message' => 'Le slug ne peut contenir que des lettres, des nombres, des tirets
        )
    );

    public function alphaNumericDashUnderscore($check) {
        // le tableau $check est passé en utilisant le nom du champ de formulaire comme cle
        // nous devons extraire la valeur pour rendre la fonction générique
        $valeur = array_values($check);
        $valeur = $valeur[0];

        return preg_match('|^[0-9a-zA-Z_-]*$|', $valeur);
    }
}
```

Note : Vos propres méthodes de validation doivent avoir une visibilité `public`. Les méthodes de Validation qui sont `protected` et `private` ne sont pas supportées.

Cette méthode devrait retourner `true` si la valeur est valide. Si la validation échoue, elle retourne `false`. L'autre valeur de retour valide est une chaîne de caractères qui sera montrée en message d'erreur. Retourner une chaîne de caractères signifie que la validation a échoué. La chaîne de caractère va surcharger le message défini dans le tableau $validate et sera montré dans le formulaire de vue comme la raison pour laquelle le champ n'est pas valide.

Changer dynamiquement les règles de validation

Utiliser la propriété `$validate` pour déclarer les règles de validation est une bonne façon de définir des règles statiques pour chaque model. Néanmoins, il y a d'autres cas où vous voudrez ajouter, modifier ou retirer dynamiquement des règles de validation d'un ensemble pré-défini.

Toutes les règles de validation sont stockées dans un objet `ModelValidator`, qui contient chaque règle pour chaque champ définie dans votre model. Définir de nouvelles règles de validation est aussi facile que de dire à cet objet de stocker de nouvelles méthodes de validation pour les champs que vous souhaitez.

Ajouter de nouvelles règles de validation

Introduit dans la version 2.2. Les objets `ModelValidator` permettent de nombreuses façons d'ajouter de nouveaux champs à définir. Le premier est l'utilisation de la méthode `add` :

```
// Dans une classe de model
$this->validator()->add('password', 'required', array(
    'rule' => 'notEmpty',
    'required' => 'create'
));
```

Cela va ajouter une règle simple au champ *password* dans le model. Vous pouvez chainer plusieurs appels à ajouter pour créer autant de règles que vous souhaitez :

```
// Dans une classe de model
$this->validator()
    ->add('password', 'required', array(
        'rule' => 'notEmpty',
        'required' => 'create'
    ))
    ->add('password', 'size', array(
        'rule' => array('between', 8, 20),
        'message' => 'Password should be at least 8 chars long'
    ));
```

Il est aussi possible d'ajouter des règles multiples en une fois pour un champ unique :

```
$this->validator()->add('password', array(
    'required' => array(
        'rule' => 'notEmpty',
        'required' => 'create'
    ),
    'size' => array(
        'rule' => array('between', 8, 20),
        'message' => 'Password should be at least 8 chars long'
    )
));
```

De façon alternative, vous pouvez utiliser l'objet validator pour définir les règles directement aux champs en utilisant l'interface de tableau :

```
$validator = $this->validator();
$validator['username'] = array(
    'unique' => array(
        'rule' => 'isUnique',
        'required' => 'create'
    ),
    'alphanumeric' => array(
        'rule' => 'alphanumeric'
    )
);
```

Modifier les règles de validation courantes

Introduit dans la version 2.2. Modifier les règles de validation courantes est aussi possible en utilisant l'objet validator, il y a plusieurs façons pour modifier les règles courantes, les méthodes d'ajout à un champ ou le retrait complet d'une règle à partir d'une règle définie d'un champ :

```
// Dans une classe de model
$this->validator()->getField('password')->setRule('required', array(
    'rule' => 'required',
    'required' => true
));
```

Vous pouvez aussi complètement remplacer toutes les règles pour un champ en utilisant une méthode similiare :

```
// Dans une classe de model
$this->validator()->getField('password')->setRules(array(
    'required' => array(...),
    'otherRule' => array(...)
));
```

Si vous souhaitez juste modifier une propriété unique dans une règle dans laquelle vous pouvez définir des propriétés directement dans l'objet `CakeValidationRule` :

```
// Dans une classe de model
$this->validator()->getField('password')
    ->getRule('required')->message = 'This field cannot be left blank';
```

Les propriétés dans toute `CakeValidationRule` sont nommées avec le tableau de clés valides que vous pouvez utiliser pour définir de telles règles en utilisant la propriété `$validate` dans le model.

Comme avec l'ajout de nouvelle règle à l'ensemble, il est aussi possible de modifier les règles existantes en utilisant l'interface de tableau :

```
$validator = $this->validator();
$validator['username']['unique'] = array(
    'rule' => 'isUnique',
    'required' => 'create'
);
```

```
$validator['username']['unique']->last = true;
$validator['username']['unique']->message = 'Name already taken';
```

Retirer des règles d'un ensemble

Introduit dans la version 2.2. Il est possible de retirer complètement toutes les règles pour un champ ou de supprimer une règle unique dans un ensemble de règles de champ :

```
// Retire complètement toutes les règles pour un champ
$this->validator()->remove('username');

// Retire la règle 'required' de password
$this->validator()->remove('password', 'required');
```

De façon optionnelle, vous pouvez utiliser l'interface de tableau pour supprimer les règles à partir d'un ensemble :

```
$validator = $this->validator();
// Retire complètement toutes les règles pour un champ
unset($validator['username']);

// Retire la règle 'required' de password
unset($validator['password']['required']);
```

Règles de validation incluses

class `Validation`

La classe de validation de CakePHP contient un certain nombre de règles prédéfinies, qui rendent la validation des données plus simple dans vos models. Cette classe contient de nombreuses règles souvent utilisées que vous n'aurez pas à ré-écrire vous même. Ci-dessous vous trouverez une liste complète de toutes les règles, illustrées par des exemples d'utilisation.

static `Validation::`**`alphaNumeric`**(*mixed $check*)

Les données pour ce champ ne doivent contenir que chiffres et lettres :

```
    public $validate = array(
        'login' => array(
            'rule'    => 'alphaNumeric',
            'message' => 'Les données pour ce champ ne doivent contenir que lettres et chi
        )
    );
```

static `Validation::`**`between`**(*string $check*, *integer $min*, *integer $max*)

La longueur des données du champ doit être comprise dans la plage numérique spécifiée. Les valeurs minimum et maximum doivent être toutes les deux fournies. Cette méthode utilise <= et non < :

```
public $validate = array(
    'mot_de_passe' => array(
        'rule'    => array('between', 5, 15),
        'message' => 'Le mot de passe doit avoir une longueur comprise entre 5 et 15 c
    )
);
```

La longueur des données est "le nombre d'octets dans la représentation des données sous forme de chaîne". Faites attention, car elle peut être plus grande que le nombre de caractères quand vous manipulez des caractères non-ASCII.

static `Validation::blank`(*mixed $check*)

Cette règle est utilisée pour vérifier que le champ est laissé vide ou que seulement des caractères blancs y sont présent. Les caractères blancs incluent l'espace, la tabulation, le retour chariot et nouvelle ligne. :

```
public $validate = array(
    'id' => array(
        'rule' => 'blank',
        'on'   => 'create'
    )
);
```

static `Validation::boolean`(*string $check*)

Les données pour ce champ doivent être une valeur booléenne. Les valeurs possibles sont : true ou false, les entiers 0 ou 1, les chaînes '0' ou '1'. :

```
public $validate = array(
    'maCaseACocher' => array(
        'rule'    => array('boolean'),
        'message' => 'Valeur incorrecte pour maCaseACocher'
    )
);
```

static `Validation::cc`(*mixed $check*, *mixed $type = 'fast'*, *boolean $deep = false*, *string $regex = null*)

Cette règle est utilisée pour vérifier si une donnée est un numéro de carte de crédit valide. Elle prend trois paramètres : 'type', 'deep' et 'regex'.

Le paramètre 'type' peut être assigné aux valeurs 'fast', 'all' ou à l'une des suivantes :
–amex
–bankcard
–diners
–disc
–electron
–enroute
–jcb
–maestro
–mc
–solo
–switch
–visa

–voyager

Si 'type' est défini à 'fast', cela valide les données de la majorité des formats numériques de cartes de crédits. Définir 'type' à 'all' vérifiera tous les types de cartes de crédits. Vous pouvez aussi définir 'type' comme un tableau des types que vous voulez détecter.

Le paramètre 'deep' devrait être défini comme une valeur booléenne. S'il est défini à true, la validation vérifiera l'algorithme Luhn de la carte de crédit (http ://en.wikipedia.org/wiki/Luhn_algorithm). Par défaut, elle est à false.

Le paramètre 'regex' vous permet de passer votre propre expression régulière, laquelle sera utilisée pour valider le numéro de la carte de crédit :

```
public $validate = array(
    'numero_cc' => array(
        'rule'    => array('cc', array('visa', 'maestro'), false, null),
        'message' => 'Le numéro de carte de crédit que vous avez saisi était invalide.
    )
);
```

static `Validation::comparison`(*mixed $check1*, *string $operator = null*, *integer $check2 = null*)

Comparison est utilisé pour comparer des valeurs numériques. Il supporte "est supérieur", "est inférieur", "supérieur ou égal", "inférieur ou égal", "égal à" et "non égal". Quelques exemples sont indiqués ci-dessous :

```
public $validate = array(
    'age' => array(
        'rule'    => array('comparison', '>=', 18),
        'message' => 'Vous devez avoir 18 ans au moins pour vous inscrire.'
    )
);

public $validate = array(
    'age' => array(
        'rule'    => array('comparison', 'greater or equal', 18),
        'message' => 'Vous devez avoir 18 ans au moins pour vous inscrire.'
    )
);
```

static `Validation::custom`(*mixed $check*, *string $regex = null*)

Utilisé quand une règle personnalisée est nécessaire :

```
public $validate = array(
    'infinite' => array(
        'rule'    => array('custom', '\u221E'),
        'message' => 'Merci de rentrer un nombre infini.'
    )
);
```

static `Validation::date`(*string $check*, *mixed $format = 'ymd'*, *string $regex = null*)

Cette règle s'assure que les données soumises sont des formats de date valides. Un seul paramètre (qui peut être un tableau) doit être passé et sera utilisé pour vérifier le format de la date soumise. La valeur de ce paramètre peut être l'une des suivantes :

–'dmy', par exemple : 27-12-2006 ou 27-12-06 (les séparateurs peuvent être l'espace, le point, le tiret, le slash)
–'mdy', par exemple : 12-27-2006 ou 12-27-06 (les séparateurs peuvent être l'espace, le point, le tiret, le slash)
–'ymd', par exemple : 2006-12-27 ou 06-12-27 (les séparateurs peuvent être l'espace, le point, le tiret, le slash)
–'dMy', par exemple : 27 Décembre 2006 ou 27 Déc 2006
–'Mdy', par exemple : Décembre 27, 2006 ou Déc 27, 2006 (la virgule est optionnelle)
–'My', par exemple : (Décembre 2006 ou Déc 2006)
–'my', par exemple : 12/2006 ou 12/06 (les séparateurs peuvent être l'espace, le point, le tiret, le slash)
–'ym' par ex. 2006/12 ou 06/12 (les séparateurs peuvent être un espace, une période, un tiret, ou un slash)
–'y' e.g. 2006 (les séparateurs peuvent être un espace, une période, un tiret, ou un slash)
Si aucune clé n'est soumise, la clé par défaut 'ymd' sera utilisée :

```
public $validate = array(
    'date_de_naissance' => array(
        'rule'       => array('date', 'ymd'),
        'message'    => 'Entrez une date valide au format AA-MM-JJ.',
        'allowEmpty' => true
    )
);
```

Etant donné que de nombreux moteurs de stockage réclament un certain format de date, vous devriez envisager de faire le plus gros du travail en acceptant un large choix de formats et en essayant de les convertir, plutôt que de forcer les gens à les soumettre dans un format donné. Le plus de travail vous ferez pour les users, le mieux ce sera. Modifié dans la version 2.4 : Les formats `ym` et `y` ont été ajoutés.

static `Validation::datetime`(*array $check*, *mixed $dateFormat = 'ymd'*, *string $regex = null*)

Cette règle s'assure que les données sont dans format datetime valide. Un paramètre (qui peut être un tableau) peut être passé pour spécifier le format de la date. La valeur du paramètre peut être une ou plusieurs des valeurs suivantes :
–'dmy', par exemple : 27-12-2006 or 27-12-06 (les séparateurs peuvent être l'espace, le point, le tiret, le slash)
–'mdy', par exemple : 12-27-2006 or 12-27-06 (les séparateurs peuvent être l'espace, le point, le tiret, le slash)
–'ymd', par exemple : 2006-12-27 or 06-12-27 (les séparateurs peuvent être l'espace, le point, le tiret, le slash)
–'dMy', par exemple : 27 December 2006 or 27 Dec 2006
–'Mdy', par exemple : December 27, 2006 or Dec 27, 2006 (le point est optionnel)
–'My', par exemple : (December 2006 or Dec 2006)
–'my', par exemple : 12/2006 or 12/06 (les séparateurs peuvent être l'espace, le point, le tiret, le slash)
Si aucune clé n'est fournie, la clé par défaut qui sera utilisée est 'ymd' :

```
public $validate = array(
    'birthday' => array(
        'rule'    => array('datetime', 'dmy'),
        'message' => 'Merci de rentrer une date et un time valide.'
```

```
        )
    );
```

Un second paramètre peut aussi être passé pour spécifier une expression réguière personnalisée. Si un paramètre est utilisé, ce sera la seule validation qui apparaitra.

Notez que au contraire de date(), datetime() validera une date et un time.

static `Validation::decimal` (*integer $check*, *integer $places = null*, *string $regex = null*)

Cette règle s'assure que la donnée est un nombre décimal valide. Un paramètre peut être passé pour spécifier le nombre de décimales requises après le point. Si aucun paramètre n'est passé, la donnée sera validée comme un nombre scientifique à virgule flottante, entraînant une erreur si aucune décimale n'est trouvée après le point :

```
    public $validate = array(
        'prix' => array(
            'rule' => array('decimal', 2)
        )
    );
```

static `Validation::email` (*string $check*, *boolean $deep = false*, *string $regex = null*)

Celle-ci vérifie que la donnée soit une adresse email valide. En passant un booléen true comme second paramètre de cette règle, elle tentera de vérifier aussi, que l'hôte de l'adresse est valide :

```
    public $validate = array('email' => array('rule' => 'email'));

    public $validate = array(
        'email' => array(
            'rule'    => array('email', true),
            'message' => 'Merci de soumettre une adresse email valide.'
        )
    );
```

static `Validation::equalTo` (*mixed $check*, *mixed $compareTo*)

Cette règle s'assurera que la valeur est égal à la valeur passée et qu'elle est du même type.

```
    public $validate = array(
        'nourriture' => array(
            'rule'    => array('equalTo', 'gâteau'),
            'message' => 'Cette valeur devrait être la chaîne gâteau'
        )
    );
```

static `Validation::extension` (*mixed $check*, *array $extensions = array('gif', 'jpeg', 'png', 'jpg')*)

Cette règle vérifie les extensions valides de fichier, comme .jpg ou .png. Permet la vérification d'extensions multiples, en les passant sous forme de tableau.

```
    public $validate = array(
        'image' => array(
            'rule'    => array('extension', array('gif', 'jpeg', 'png', 'jpg')),
            'message' => 'Merci de soumettre une image valide.'
        )
    );
```

static `Validation::fileSize` (*$check*, *$operator = null*, *$size = null*)

Cette règle vous permet de vérifier les tailles de fichier. Vous pouvez utiliser `$operator` pour décider du type de comparaison que vous souhaitez utiliser. Tous les opérateurs supportés par `comparison()` sont ici aussi supportés. Cette méthode va gérer automatiquement les tableaux de valeur à partir de `$_FILES` en lisant la clé `tmp_name` si `$check` est un tableau qui contient cette clé :

```
public $validate = array(
    'image' => array(
        'rule' => array('fileSize', '<=', '1MB'),
        'message' => 'L\'Image doit être inférieur à 1MB'
    )
);
```

Introduit dans la version 2.3 : Cette méthode a été ajoutée dans 2.3

static `Validation::inList` (*string $check*, *array $list*)

Cette règle s'assurera que la valeur est dans un ensemble donné. Elle nécessite un tableau des valeurs. Le champ est valide si sa valeur vérifie l'une des valeurs du tableau donné.

Exemple :

```
public $validate = array(
    'fonction' => array(
         'choixAutorise' => array(
             'rule'    => array('inList', array('Foo', 'Bar')),
             'message' => 'Entrez soit Foo, soit Bar.'
         )
     )
 );
```

static `Validation::ip` (*string $check*, *string $type = 'both'*)

Cette règle s'assurera qu'une adresse IPv4 ou IPv6 valide ait été soumise. Accepte 'both' en option (par défaut), 'IPv4' ou 'IPv6'.

```
public $validate = array(
    'ip_client' => array(
        'rule'    => array('ip', 'IPv4'), // ou 'IPv6' ou 'both' (par défaut)
        'message' => 'Merci de soumettre une adresse IP valide.'
    )
);
```

`Model::isUnique()`

La donnée pour le champ doit être unique, elle ne peut être utilisée par aucune autre ligne.

```
public $validate = array(
    'login' => array(
        'rule'    => 'isUnique',
        'message' => 'Ce nom d\'user a déjà été choisi.'
    )
);
```

static `Validation::luhn` (*string|array $check*, *boolean $deep = false*)
L'algorithme Luhn : Une formule de vérification de somme pour valider un ensemble de nombres d'identification. Regardez http ://en.wikipedia.org/wiki/Luhn_algorithm pour plus d'informations.

static `Validation::maxLength` (*string $check*, *integer $max*)
Cette règle s'assure que la donnée respecte la longueur maximale requise.

```
public $validate = array(
    'login' => array(
        'rule'    => array('maxLength', 15),
        'message' => 'Les noms d\'user ne doivent pas dépasser 15 caractères.'
    )
);
```

La longueur ici est "le nombre de bytes dans la représentation en chaîne de la donnée". Faites attention car elle pourrait être plus grande que le nombre de caractères en manipulant des caractères non-ASCII.

static `Validation::mimeType` (*mixed $check*, *array $mimeTypes*)
Introduit dans la version 2.2. Cette règle vérifie la validité d'un mimeType.

```
public $validate = array(
    'image' => array(
        'rule'    => array('mimeType', array('image/gif')),
        'message' => 'Invalid mime type.'
    ),
);
```

static `Validation::minLength` (*string $check*, *integer $min*)
Cette règle s'assure que les données ont une obligation de longueur minimum.

```
public $validate = array(
    'login' => array(
        'rule'    => array('minLength', 8),
        'message' => 'Usernames must be at least 8 characters long.'
    )
);
```

La longueur ici est "le nombre de bytes dans la représentation des données sous forme de chaîne". Faites attention car elle pourrait être plus grande que le nombre de caractères en manipulant des caractères non-ASCII.

static `Validation::money` (*string $check*, *string $symbolPosition = 'left'*)
Cette règle s'assurera que la valeur est une somme monétaire valide.

Le second paramètre définit où le symbole est situé (gauche/droite).

```
public $validate = array(
    'salaire' => array(
        'rule'    => array('money', 'left'),
        'message' => 'Merci de soumettre une somme monétaire valide.'
    )
);
```

static `Validation::multiple` (*mixed $check*, *mixed $options = array()*)

Utilisez cette règle pour valider un champ select multiple. Elle accepte les paramètres "in", "max" et "min".

```
public $validate = array(
    'multiple' => array(
        'rule' => array('multiple', array(
            'in'  => array('do', 'ré', 'mi', 'fa', 'sol', 'la', 'si'),
            'min' => 1,
            'max' => 3
        )),
        'message' => 'Merci de choisir une, deux ou trois options'
    )
);
```

static `Validation::notEmpty` (*mixed $check*)

La règle de base pour s'assurer qu'un champ n'est pas vide. :

```
public $validate = array(
    'titre' => array(
        'rule'    => 'notEmpty',
        'message' => 'Ce champ ne peut pas rester vide'
    )
);
```

Ne l'utilisez pas pour un champ select multiple, sinon cela causera une erreur. A la place, utilisez "multiple".

static `Validation::numeric` (*string $check*)

Vérifie si la donnée passée est un nombre valide. :

```
public $validate = array(
    'cars' => array(
        'rule'    => 'numeric',
        'message' => 'Merci de soumettre le nombre de voitures.'
    )
);
```

static `Validation::naturalNumber` (*mixed $check*, *boolean $allowZero = false*)

Introduit dans la version 2.2. Cette règle vérifie si une donnée passée est un nombre entier naturel valide. Si `$allowZero` est défini à true, la valeur zero est aussi acceptée.

```
public $validate = array(
    'wheels' => array(
        'rule'    => 'naturalNumber',
        'message' => 'Merci de fournir le nombre de pneus.'
    ),
    'airbags' => array(
        'rule'    => array('naturalNumber', true),
        'message' => 'Merci de remplir le nombre d'airbags.'
    ),
);
```

static `Validation::phone` (*mixed $check*, *string $regex = null*, *string $country = 'all'*)

Phone valide les numéros de téléphone US. Si vous voulez valider des numéros de téléphones non-US, vous pouvez fournir une expression régulière comme second paramètre pour couvrir des formats de numéros supplémentaires.

```
public $validate = array(
    'telephone' => array(
        'rule' => array('phone', null, 'us')
    )
);
```

static `Validation::postal` (*mixed $check*, *string $regex = null*, *string $country = 'us'*)

Postal est utilisé pour valider des codes postaux des U.S.A. (us), du Canada (ca), du Royaume-Uni (uk), de l'Italie (it), d'Allemagne (de) et de Belgique (be). Pour les autres formats de codes postaux, vous devez fournir une expression régulière comme second paramètre.

```
public $validate = array(
    'code_postal' => array(
        'rule' => array('postal', null, 'us')
    )
);
```

static `Validation::range` (*string $check*, *integer $lower = null*, *integer $upper = null*)

Cette règle s'assure que la valeur est dans une fourchette donnée. Si aucune fourchette n'est soumise, la règle s'assurera que la valeur est un nombre limite valide pour la plateforme courante.

```
public $validate = array(
    'nombre' => array(
        'rule'    => array('range', -1, 11),
        'message' => 'Merci d\'entrer un nombre entre 0 et 10'
    )
);
```

L'exemple ci-dessus acceptera toutes les valeurs qui sont plus grandes que 0 (par ex, 0.01) et plus petite que 10 (par ex, 9.99). Note : Les deux extrémités données (-1 et 11) ne sont pas incluses ! ! !

Note : The range lower/upper are not inclusive

---

static `Validation::ssn` (*mixed $check*, *string $regex = null*, *string $country = null*)

Ssn valide les numéros de sécurité sociale des U.S.A. (us), du Danemark (dk) et des Pays-Bas (nl). Pour les autres formats de numéros de sécurité sociale, vous devez fournir une expression régulière.

```
public $validate = array(
    'ssn' => array(
        'rule' => array('ssn', null, 'us')
    )
);
```

static `Validation::time` (*string $check*)

La validation du Time, détermine si une chaîne de caractères est un time valide. Valide le time en 24hr (HH :MM) ou am/pm ([H]H :MM[a|p]m). N'autorise/ne valide pas les secondes.

static `Validation::uploadError` (*mixed $check*)

Introduit dans la version 2.2. Cet règle vérifie si un upload de fichier connait une erreur.

```
public $validate = array(
    'image' => array(
        'rule'    => 'uploadError',
        'message' => 'Something went wrong with the upload.'
    ),
);
```

static `Validation::url` (*string $check*, *boolean $strict = false*)

Cette règle vérifie les formats valides d'URL. Elle supporte les protocoles http(s), ftp(s), file, news et gopher :

```
public $validate = array(
    'siteweb' => array(
        'rule' => 'url'
    )
);
```

Pour s'assurer qu'un protocole est présent dans l'url, le mode strict mode peut être activé comme ceci :

```
public $validate = array(
    'siteweb' => array(
        'rule' => array('url', true)
    )
);
```

static `Validation::userDefined` (*mixed $check*, *object $object*, *string $method*, *array $args = null*)

Lance une validation de définition d'user.

static `Validation::uuid` (*string $check*)

Vérifie que la valeur est une valeur uuid valide : http ://tools.ietf.org/html/rfc4122

Validation localisée

Les règles de validation phone() et postal() vont envoyer les préfixes de pays qu'elles ne savent pas gérer à une autre classe avec le nom afférant. Par exemple si vous vivez aux Pays-Bas, vous pourriez créer une classe comme :

```
class NlValidation {
    public static function phone($check) {
        // ...
    }
    public static function postal($check) {
        // ...
    }
}
```

Ce fichier pourra être placé dans `APP/Validation/` ou `App/PluginName/Validation/`, mais doit être importé via App : :uses() avant tout tentative d'utilisation. Dans votre validation de model, vous pourrez utiliser votre classe NlValidation en faisant ce qui suit :

```
public $validate = array(
    'phone_no' => array('rule' => array('phone', null, 'nl')),
    'postal_code' => array('rule' => array('postal', null, 'nl')),
);
```

Quand vos données de model sont validées, la Validation va voir qu'elle ne peut pas gérer la locale `nl` et va tenter de déléguer à `NlValidation::postal()` et le retour de cette méthode va être utilisée comme réussite/echec pour la validation. Cette approche vous permet de créer des classes qui gèrent un sous-ensemble ou groupe de locales, chose qu'un large switch ne pourrait pas faire. L'utilisation des méthodes de validation individuelle n'a pas changé, la possibilité de faire passer à un autre validateur a été ajouté.

Astuce : Le Plugin Localized contient déjà beaucoup de règles prêtes à être utilisées : https ://github.com/cakephp/localized Aussi n'hésitez pas à contribuer en donnant vos règles de validation localisées.

Validation des données à partir du Controller

Alors que normalement vous n'utiliseriez que la méthode save du model, il peut arriver que vous souhaitiez valider les données sans les sauvegarder. Par exemple, vous souhaitez afficher des informations supplémentaires à l'utilisateur avant qu'il ne sauvegarde les données dans la base. Valider les données nécessite un processus légèrement différent de la méthode save.

Tout d'abord, mettez les données dans le model :

```
$this->ModelName->set($this->request->data);
```

Ensuite, pour vérifier si les données sont validées, utilisez la méthide validates du model, qui va retourner true si elles sont valides et false si elles ne le sont pas :

```
if ($this->ModelName->validates()) {
    // La logique est validée
} else {
    // La logique n'est pas validée
    $errors = $this->ModelName->validationErrors;
}
```

Il peut être souhaité de valider votre model seulement en utilisant un sous-ensemble des validations spécifiées dans le model. Par exemple, si vous avez un model Utilisateur avec les champs prenom, nom, email et mot_de_passe. Dans ce cas, quand vous créez ou modifiez un utilisateur, vous ne voulez pas valider les 4 règles des champs. Pourtant quant un utilisateur se connecte, vous voulez validez seulement les règles de l'email et du mot_de_passe. Pour le faire, vous pouvez passer un tableau d'options spécifiant les champs sur lesquels vous voulez la validation. Par exemple

```
if ($this->User->validates(array('fieldList' => array('email', 'mot_de_passe'))) {
    // valide
} else {
    // invalide
}
```

La méthode validates invoque la méthode invalidFields qui remplit la propriété validationErrors du model. La méthode invalidFields retourne aussi cette donnée comme résultat :

```
$errors = $this->ModelName->invalidFields(); // contient le tableau des ErreursDeValidatio
```

La liste des erreurs de validation n'est pas supprimée entre les différents appels à `invalidFields()`. Donc si vous validez dans une boucle et que vous voulez chaque jeu d'erreurs séparement, n'utilisez pas `invalidFields()`. Utilisez plutôt `validates()` et accéder à la propriété `validationErrors` du model.

Il est important de noter que les données doivent être envoyées au model avant que les données soient validées. C'est différent de la méthode save qui autorise aux données d'être passées comme paramètre. Aussi, garder à l'esprit qu'il n'est pas requis d'appeler validates antérieurement à l'appel save puisque save va automatiquement valider les données avant l'enregistrement effectif.

Pour valider plusieurs models, l'approche suivante devrait être utilisée :

```
if ($this->ModelName->saveAll($this->request->data, array('validate' => 'only'))) {
  // valide
} else {
  // ne valide pas
}
```

Si vous avez validé les données avant l'enregistrement, vous pouvez stopper la validation du save pour éviter un deuxième contrôle :

```
if ($this->ModelName->saveAll($this->request->data, array('validate' => false))) {
    // sauvegarder sans validation
}
```

Méthodes Callback

Si vous voulez glisser un bout de logique applicative juste avant ou après une opération d'un model CakePHP, utilisez les callbacks de model. Ces fonctions peuvent être définies dans les classes de model (cela comprend également votre classe AppModel). Notez bien les valeurs de retour attendues pour chacune de ces méthodes spéciales.

Lors de l'utilisation de méthodes de callback, vous devriez vous rappeler que les callbacks des behaviors sont lancés **avant** les callbacks des models.

beforeFind

`beforeFind(array $queryData)`

Appelée avant toute opération liée à la recherche. Les `$queryData` passées à cette méthode de callback contiennent des informations sur la requête courante : conditions, champs, etc.

Si vous ne souhaitez pas que l'opération de recherche commence (par rapport à une décision liée aux options de `$queryData`), retournez *false*. Autrement, retournez la variable `$queryData` éventuellement modifiée, ou tout ce que vous souhaitez voir passé à la méthode find() ou ses équivalents.

Vous pouvez utiliser cette méthode de callback pour restreindre les opérations de recherche en se basant sur le rôle de l'utilisateur, ou prendre des décisions sur la politique de mise en cache en fonction de la charge actuelle.

afterFind

`afterFind(array $results, boolean $primary = false)`

Utilisez cette méthode de callback pour modifier les résultats qui ont été retournés par une opération de recherche, ou pour effectuer toute logique post-recherche. Le paramètre $results passé à cette méthode contient les résultats retournés par l'opération find() du model, càd quelque chose comme :

```
$results = array(
    0 => array(
        'NomModel' => array(
            'champ1' => 'valeur1',
            'champ2' => 'valeur2',
        ),
    ),
);
```

La valeur de retour de ce callback doit être le résultat de l'opération de recherche (potentiellement modifié) qui a déclenché ce callback.

Le paramètre `$primary` indique si oui ou non le model courant est le model d'où la requête provenait ou non en tant qu'association. Si un model est requêté en tant qu'association, le format de `$results` peut différer ; à la place du résultat, que vous auriez normalement obtenu à partir d'une opération find, vous obtiendriez peut-être ça :

```
$results = array(
    'champ1' => 'valeur1',
    'champ2' => 'valeur2'
);
```

> **Warning :** Un code nécessitant que `$primaire` soit vrai aura probablement l'erreur fatale "Cannot use string offset as an array" de la part de PHP si une recherche récursive est utilisée.

Ci-dessous un exemple de la manière dont afterfind peut être utilisée pour formater des dates :

```
public function afterFind($results, $primary = false) {
    foreach ($results as $key => $val) {
        if (isset($val['Event']['begindate'])) {
            $results[$key]['Event']['begindate'] = $this->dateFormatAfterFind($val['Event']
        }
```

```
    }
    return $results;
}

public function dateFormatAfterFind($dateString) {
    return date('d-m-Y', strtotime($dateString));
}
```

beforeValidate

`beforeValidate(array $options = array())`

Utilisez ce rappel pour modifier les données du model avant qu'elles ne soient validées ou pour modifier les règles de validation si nécessaire. Cette fonction doit aussi retourner *vrai*, sinon l'exécution du save() courant sera annulée.

beforeSave

`beforeSave(array $options = array())`

Placez toute logique de pré-enregistrement dans cette fonction. Cette fonction s'exécute immediatement après que les données du model ont été validées avec succès, mais juste avant que les données ne soient sauvegardées. Cette fonction devrait toujours retourner vrai si voulez que l'opération d'enregistrement se poursuive.

Ce callback est particulièrement pratique, pour toute logique de manipulation des données qui nécessite de se produire avant que vos données ne soient stockées. Si votre moteur de stockage nécessite un format spécifique pour les dates, accédez-y par $this->data et modifiez-les.

Ci-dessous un exemple montrant comment beforeSave peut-être utilisé pour la conversion de date. Le code de l'exemple est utilisé pour une application qui a une date de début, au format YYYY-MM-DD dans la base de données et au format DD-MM-YYYY dans l'affichage de l'application. Bien sûr, ceci peut être très facilement modifié. Utilisez le code ci-dessous dans le model approprié.

```
public function beforeSave($options = array()) {
    if (!empty($this->data['Event']['begindate']) && !empty($this->data['Event']['enddate']
        $this->data['Event']['begindate'] = $this->dateFormatBeforeSave($this->data['Event'
        $this->data['Event']['enddate'] = $this->dateFormatBeforeSave($this->data['Event']
    }
    return true;
}

public function dateFormatBeforeSave($dateString) {
    return date('Y-m-d', strtotime($dateString));
}
```

Astuce : Assurez-vous que beforeSave() retourne vrai ou bien votre sauvegarde échouera.

afterSave

`afterSave(boolean $created)`

Si vous avez besoin d'exécuter de la logique juste après chaque opération de sauvegarde, placez-la dans cette méthode de rappel.

La valeur de `$created` sera vrai si un nouvel objet a été créé (plutôt qu'un objet mis à jour).

beforeDelete

`beforeDelete(boolean $cascade = true)`

Placez dans cette fonction, toute logique de pré-suppression. Cette fonction doit retourner vrai si vous voulez que la suppression continue et faux si vous voulez l'annuler.

La valeur de `$cascade` sera `true`, pour que les enregistrements qui dépendent de cet enregistrement soient aussi supprimés.

Astuce : Assurez vous que beforeDelete() retourne true, ou votre suppression ne va pas marcher.

```
// using app/Model/ProduitCategory.php
// Dans l'exemple suivant, ne laissez pas une catégorie être supprimée si elle contient des
// Un appel de $this->Produit->delete($id) de ProduitsController.php a défini $this->id .
// En admettant que 'ProduitCategory hasMany Produit', nous pouvons accéder à $this->Produi
public function beforeDelete() {
    $count = $this->Product->find("count", array(
        "conditions" => array("produit_category_id" => $this->id)
    ));
    if ($count == 0) {
        return true;
    } else {
        return false;
    }
}
```

afterDelete

`afterDelete()`

Placez dans cette méthode de rappel, toute logique que vous souhaitez exécuter après chaque suppression.

onError

`onError()`

Appelée si il se produit quelque problème que ce soit.

Behaviors (Comportements)

Les behaviors (comportements) de Model sont une manière d'organiser certaines des fonctionnalités définies dans les models CakePHP. Ils nous permettent de séparer la logique qui ne doit pas être directement reliée à un model, mais qui nécessite d'être là. En offrant une simple, mais puissante, manière d'étendre les models, les behaviors nous permettent d'attacher des fonctionnalités aux models en définissant une simple variable de classe. C'est comme çà que les behaviors permettent de débarrasser les models de tout le "sur-poids" qui ne devrait pas faire partie du contrat métier qu'ils modèlent ou de ce qui est aussi nécessité par différents models et qui peut alors être extrapolé.

Par exemple, considérez un model qui nous donne accès à une table qui stocke des informations sur la structure d'un arbre hiérarchique. Supprimer, ajouter ou déplacer les nœuds dans l'arbre n'est pas aussi simple que d'effacer, d'insérer ou d'éditer les lignes d'une table. De nombreux enregistrements peuvent nécessiter une mise à jour suite au déplacement des éléments. Plutôt que de créer ces méthodes de manipulation d'arbre une fois par model de base (pour chaque model nécessitant cette fonctionnalité), nous pourrions simplement dire à notre model d'utiliser le behavior `TreeBehavior` ou, en des termes plus formels, nous dirions à notre model de se comporter comme un Arbre. On appelle cela attacher un behavior à un model. Avec une seule ligne de code, notre model CakePHP disposera d'un nouvel ensemble complet de méthodes lui permettant d'interagir avec la structure sous-jacente.

CakePHP contient déjà des behaviors pour les structures en arbre, les contenus traduits, les interactions par liste de contrôle d'accès, sans oublier les behaviors des contributeurs de la communauté déjà disponibles dans la Boulangerie (Bakery) CakePHP (http ://bakery.cakephp.org). Dans cette section nous couvrirons le schéma d'usage classique pour ajouter des behaviors aux models, l'utilisation des behaviors intégrés à CakePHP et la manière de créer nos propres behaviors.

Au final, les Behaviors sont Mixins (http ://en.wikipedia.org/wiki/Mixin) avec les callbacks.

Utiliser les Behaviors

Les Behaviors sont attachés aux models grâce à la variable `$actsAs` des classes model :

```
class Category extends AppModel {
    public $actsAs = array('Tree');
}
```

Cette exemple montre comme un model Category pourrait être gérer dans une structure en arbre en utilisant le behavior Tree. Une fois qu'un behavior a été spécifié, utilisez les méthodes qu'il ajoute comme si elles avaient toujours existé et fait partie du model original :

```
// Définir ID
$this->Category->id = 42;

// Utiliser la méthode children() du behavior:
$kids = $this->Category->children();
```

Quelques behaviors peuvent nécessiter ou permettre des réglages quand ils sont attachés au model. Ici, nous indiquons à notre behavior Tree les noms des champs "left" et "right" de la table sous-jacente :

```
class Category extends AppModel {
    public $actsAs = array('Tree' => array(
        'left'  => 'left_node',
        'right' => 'right_node'
    ));
}
```

Nous pouvons aussi attacher plusieurs behaviors à un model. Il n'y aucune raison pour que, par exemple, notre model Category se comporte seulement comme un arbre, il pourrait aussi supporter l'internationalisation :

```
class Category extends AppModel {
    public $actsAs = array(
        'Tree' => array(
          'left'  => 'left_node',
          'right' => 'right_node'
        ),
        'Translate'
    );
}
```

Jusqu'à présent, nous avons ajouter les behaviors aux models en utilisant une variable de classe. Cela signifie que nos behaviors seront attachés à nos models de tout au long de leur durée vie. Pourtant, nous pourrions avoir besoin de "détacher" les behaviors des models à l'exécution. Considérons que dans notre précédent model Category, lequel agit comme un model Tree et Translate, nous ayons besoin pour quelque raison de le forcer à ne plus agir comme un model Translate :

```
// Détache un behavior de notre model :
$this->Category->Behaviors->unload('Translate');
```

Cela fera que notre model Categorie arrêtera dorénavant de se comporter comme un model Translate. Nous pourrions avoir besoin, sinon, de désactiver simplement le behavior Translate pour qu'il n'agisse pas sur les opérations normales de notre model : nos finds, nos saves, etc. En fait, nous cherchons à désactiver le behavior qui agit sur nos callbacks de model CakePHP. Au lieu de détacher le behavior, nous allons dire à notre model d'arrêter d'informer ses callbacks du behavior Translate :

```
// Empêcher le behavior de manipuler nos callbacks de model
$this->Category->Behaviors->disable('Translate');
```

Nous pourrions également avoir besoin de chercher si notre behavior manipule ces callbacks de model et si ce n'est pas le cas, alors de restaurer sa capacité à réagir avec eux :

```
// Si notre behavior ne manipule pas nos callbacks de model
if (!$this->Category->Behaviors->enabled('Translate')) {
    // Disons lui de le faire maintenant !
    $this->Category->Behaviors->enable('Translate');
}
```

De la même manière que nous pouvons détacher complètement un behavior d'un model à l'exécution, nous pouvons aussi attacher de nouveaux behaviors. Disons que notre model familier Category nécessite de se comporter comme un model Christmas, mais seulement le jour de Noël :

```
// Si nous sommes le 25 déc
if (date('m/d') === '12/25') {
    // Notre model nécessite de se comporter comme un model Christmas
    $this->Category->Behaviors->load('Christmas');
}
```

Nous pouvons aussi utiliser la méthode attach pour surcharger les réglages du behavior :

```
// Nous changerons un réglage de notre behavior déjà attaché
$this->Category->Behaviors->load('Tree', array('left' => 'new_left_node'));
```

Il y a aussi une méthode pour obtenir la liste des behaviors qui sont attachés à un model. Si nous passons le nom d'un behavior à une méthode, elle nous dira si ce behavior est attaché au model, sinon elle nous donnera la liste des behaviors attachés :

```
// Si le behavior Translate n'est pas attaché
if (!$this->Category->Behaviors->attached('Translate')) {
    // Obtenir la liste de tous les behaviors qui sont attachés au model
    $behaviors = $this->Category->Behaviors->attached();
}
```

Créer des Behaviors

Les behaviors qui sont attachés aux Models voient leurs callbacks appelés automatiquement. Ces callbacks sont similaires à ceux qu'on trouve dans les Models : `beforeFind`, `afterFind`, `beforeSave`, `afterSave`, `beforeDelete`, `afterDelete` et `onError`. Voir *Méthodes Callback*.

Vos behaviors devront être placés dans `app/Model/Behavior`. Ils sont nommés en CamelCase et suffixés par `Behavior`, par ex. NomBehavior.php. Il est utile d'utiliser un behavior du coeur comme template quand on crée son propre behavior. Vous les trouverez dans `lib/Cake/Model/Behavior/`.

Chaque callback et behavior prend comme premier paramètre, une référence du model par lequel il est appelé.

En plus de l'implémentation des callbacks, vous pouvez ajouter des réglages par behavior et/ou par liaison d'un behavior au model. Des informations à propos des réglages spécifiques peuvent être trouvées dans les chapitres concernant les behaviors du cœur et leur configuration.

Voici un exemple rapide qui illustre comment les réglages peuvent êtres passés du model au behavior :

```
class Post extends AppModel {
    public $actsAs = array(
        'YourBehavior' => array(
            'option1_key' => 'option1_valeur'
        )
    );
}
```

Puisque les behaviors sont partagés à travers toutes les instances de model qui l'utilisent, une bonne pratique pour stocker les paramètres par nom d'alias/model qui utilise le behavior. La création des behaviors entraînera l'appel de leur méthode `setup()` :

```
public function setup(Model $Model, $settings = array()) {
    if (!isset($this->settings[$Model->alias])) {
        $this->settings[$Model->alias] = array(
            'option1_key' => 'option1_default_value',
            'option2_key' => 'option2_default_value',
            'option3_key' => 'option3_default_value',
        );
    }
    $this->settings[$Model->alias] = array_merge(
        $this->settings[$Model->alias], (array)$settings);
}
```

Créer les méthodes du behavior

Les méthodes du Behavior sont automatiquement disponibles sur tout model qui 'act as' le behavior. Par exemple si vous avez :

```
class Duck extends AppModel {
    public $actsAs = array('Flying');
}
```

Vous seriez capable d'appeler les méthodes de `FlyingBehavior` comme si elles étaient des méthodes du model Duck. Quand on créer des méthodes d'un behavior, vous obtenez automatiquement une référence du model appelé en premier paramètre. Tous les autres paramètres fournis sont shifté one place to the right. Par exemple :

```
$this->Duck->fly('toronto', 'montreal');
```

Bien que cette méthode prenne deux paramètres, la méthode signature ressemblerait à cela :

```
public function fly(Model $Model, $from, $to) {
    // Faire quelque chose à la volée.
}
```

Gardez à l'esprit que les méthodes appelées dans un fashion `$this->doIt()` à partir de l'intérieur d'une méthode d'un behavior n'obtiendra pas le paramètre $model automatiquement annexé.

Méthodes mappées

En plus de fournir des méthodes 'mixin', les behaviors peuvent aussi fournir des méthodes d'appariemment de formes (pattern matching). Les Behaviors peuvent aussi définir des méthodes mappées. Les méthodes mappées utilisent les pattern matching for method invocation. Cela vous permet de créer des méthodes du type `Model::findAllByXXX` sur vos behaviors. Les méthodes mappées ont besoin d'être déclarées dans votre tableau `$mapMethods` de behaviors. La signature de la méthode pour une méthode mappée est légèrement différente de celle d'une méthode mixin normal d'un behavior :

```
class MyBehavior extends ModelBehavior {
    public $mapMethods = array('/do(\w+)/' => 'doSomething');

    public function doSomething($model, $method, $arg1, $arg2) {
        debug(func_get_args());
        //faire quelque chose
    }
}
```

Ce qui est au-dessus mappera chaque méthode `doXXX()` appélé vers le behavior. Comme vous pouvez le voir, le model est toujours le premier paramètre, mais le nom de la méthode appelée sera le deuxième paramètre. Cela vous permet de munge le nom de la méthode pour des informations supplémentaires, un peu comme `Model::findAllByXX`. Si le behavior du dessus est attaché à un model, ce qui suit arrivera :

```
$model->doReleaseTheHounds('homer', 'lenny');

// sortira
'ReleaseTheHounds', 'homer', 'lenny'
```

Callbacks du Behavior

Les Behaviors d'un Model peuvent définir un nombre de callbacks qui sont déclenchés before les callbacks du model du même nom. Les callbacks du Behavior vous permettent de capturer des évènements dans les models attachés et d'augmenter les paramètres ou accoler dans un behavior supplémentaire.

Les callbacks disponibles sont :
- `beforeValidate`
- `beforeFind`
- `afterFind`
- `beforeSave`
- `afterSave`
- `beforeDelete`
- `afterDelete`

Créer un callback du behavior

class `ModelBehavior`

Les callbacks d'un behavior d'un model sont définis comme de simples méthodes dans votre classe de behavior. Un peu comme les méthodes classiques du behavior, ils reçoivent un paramètre `$Model` en premier argument. Ce paramètre est le model pour lequel la méthode du behavior a été invoquée.

`ModelBehavior::`**`setup`**(*Model $Model*, *array $settings = array()*)
Appelé quand un behavior est attaché à un model. Les paramètres viennent de la propriété `$actsAs` du model attaché.

`ModelBehavior::`**`cleanup`**(*Model $Model*)
Appelé quand un behavior est détaché d'un model. La méthode de base retire les paramètres du model

basées sur `$model->alias`. Vous pouvez écraser cette méthode et fournir une fonctionnalité personnalisée nettoyée.

`ModelBehavior::`**`beforeFind`**(*Model $Model*, *array $query*)
Si le beforeFind du behavior retourne false, cela annulera le find(). Retourner un tableau augmentera les paramètres de requête utilisés pour l'opération find.

`ModelBehavior::`**`afterFind`**(*Model $Model*, *mixed $results*, *boolean $primary*)
Vous pouvez utiliser le afterFind pour augmenter les résultats d'un find. La valeur retournée sera passée en résultats soit au behavior suivant dans la chaîne, soit au afterFind du model.

`ModelBehavior::`**`beforeDelete`**(*Model $Model*, *boolean $cascade = true*)
Vous pouvez retourner false d'un beforeDelete d'un behavior pour annuler la suppression. Retourne true pour autoriser la suite.

`ModelBehavior::`**`afterDelete`**(*Model $Model*)
Vous pouvez utiliser afterDelete pour effectuer des opérations de nettoyage liées à votre behavior.

`ModelBehavior::`**`beforeSave`**(*Model $Model*)
Vous pouvez retourner false d'un beforeSave d'un behavior pour annuler la sauvegarde. Retourner true pour permettre de continuer.

`ModelBehavior::`**`afterSave`**(*Model $Model*, *boolean $created*)
Vous pouvez utiliser afterSave pour effectuer des opérations de nettoyage liées au behavior. $created sera à true quand un enregistrement sera crée, et à false quand un enregistrement sera mis à jour.

`ModelBehavior::`**`beforeValidate`**(*Model $Model*)
Vous pouvez utiliser beforeValidate pour modifier un tableau de validation de model ou gérer tout autrre logique de pré-validation. Retourner false d'un callback beforeValidate annulera la validation et entraînera son echec.

DataSources (Sources de Données)

Les Sources de données (DataSources) sont les liens entre les models et la source de données qu'ils représentent. Dans de nombreux cas, les données sont récupérées depuis une base de données relationnelle telle MySQL, PostgreSQL ou MSSQL. CakePHP est distribué avec de nombreuses sources de données spécifiques d'une base de données (voir les fichiers de classe dans `lib/Cake/Model/Datasource/Database`), un résumé de ceux-ci est listé ici pour votre confort :
- MySql
- Postgres
- Sqlite
- Sqlserver

Note : Vous pouvez trouver des sources de données contributives de la communauté supplémentaires dans le Dépôt de Sources de Données CakePHP sur github (https ://github.com/cakephp/datasources/tree/2.0).

Quand vous spécifiez une configuration de connexion à une base de données dans `app/Config/database.php`, CakePHP utilise de manière transparente la source de données correspondant à la base de données pour toutes les opérations de model. Donc, même si vous pensiez ne rien connaître aux sources de données, vous les utilisez tout le temps.

Toutes les sources ci-dessus dérivent d'une classe de base `DboSource`, qui agrège de la logique commune à la plupart des bases de données relationnelles. Si vous décidez d'écrire une source de donnée RDBMS, travailler à partir de l'une d'entre elles (par ex Mysql ou Sqlite) est plus sûr.

La plupart des gens cependant, sont intéressés par l'écriture de sources de données pour des sources externes, telles les APIs REST distantes ou même un serveur LDAP. C'est donc ce que nous allons voir maintenant.

API basique pour les Sources de Données

Une source de données peut et *devrait* implémenter au moins l'une des méthodes suivantes : `create`, `read`, `update` et/ou `delete` (les signatures exactes de méthode et les détails d'implémentation ne sont pas importants pour le moment, ils seront décrits plus tard). Vous n'êtes pas obligé d'implémenter plus que nécessaire, parmi les méthodes listées ci-dessus - si vous avez besoin d'une source de données en lecture seule, il n'y a aucune raison d'implémenter `create`, `update` et `delete`.

Méthodes qui doivent être implémentées pour toutes les méthodes CRUD :
- `describe($model)`
- `listSources($data = null)`
- `calculate($model, $func, $params)`
- Au moins une des suivantes :
  - `create(Model $model, $fields = null, $values = null)`
  - `read(Model $model, $queryData = array(), $recursive = null)`
  - `update(Model $model, $fields = null, $values = null, $conditions = null)`
  - `delete(Model $model, $id = null)`

Il est possible également (et souvent très pratique), de définir l'attribut de classe `$_schema` au sein de la source de données elle-même, plutôt que dans le model.

Et c'est à peu près tout ce qu'il y a dire ici. En couplant cette source de données à un model, vous êtes alors en mesure d'utiliser `Model::find()/save()/delete()`, comme vous le feriez normalement ; les données et/ou paramètres appropriés, utilisés pour appeler ces méthodes, seront passés à la source de données elle-même, dans laquelle vous pouvez décider d'implémenter toutes les fonctionnalités dont vous avez besoin (par exemple les options de Model : :find comme le parsing `'conditions'`, `'limit'` ou même vos paramètres personnalisés).

Un Exemple

Une des raisons pour laquelle vous voudriez écrire votre propre source de données pourrait être la volonté d'accéder à l'API d'une librairie tierce en utilisant les méthodes habituelles `Model::find()/save()/delete()`. Ecrivons une source de données qui va accéder à une API JSON distante et fictive. Nous allons l'appeler `FarAwaySource` et nous allons la placer dans `app/Model/Datasource/FarAwaySource.php` :

```
App::uses('HttpSocket', 'Network/Http');

class FarAwaySource extends DataSource {

/**
```

```
 * Une description optionnelle de votre source de données
 */
    public $description = 'A far away datasource';

/**
 * Nos options de config par défaut. Ces options seront personnalisées dans notre
 * ``app/Config/database.php`` et seront fusionnées dans le ``__construct()``.
 */
    public $config = array(
        'apiKey' => '',
    );

/**
 * Si nous voulons create() ou update(), nous avons besoin de spécifier la
 * disponibilité des champs. Nous utilisons le même tableau indicé comme nous le faisions
 * fixtures et schema de migrations.
 */
    protected $_schema = array(
        'id' => array(
            'type' => 'integer',
            'null' => false,
            'key' => 'primary',
            'length' => 11,
        ),
        'name' => array(
            'type' => 'string',
            'null' => true,
            'length' => 255,
        ),
        'message' => array(
            'type' => 'text',
            'null' => true,
        ),
    );

/**
 * Créons notre HttpSocket et gérons any config tweaks.
 */
    public function __construct($config) {
        parent::__construct($config);
        $this->Http = new HttpSocket();
    }

/**
 * Puisque les sources de données se connectent normalement à une base de données
 * il y a quelques modifications à faire pour les faire marcher sans base de données.
 */

/**
 * listSources() est pour la mise en cache. Vous voulez implémenter la mise en cache
 * de votre façon avec une source de données personnalisée. Donc juste ``return null``.
 */
    public function listSources() {
```

```
        return null;
    }

/**
 * describe() dit au model votre schema pour ``Model::save()``.
 *
 * Vous voulez peut-être un schema différent pour chaque model mais utiliser
 * toujours une unique source de données. Si c'est votre cas, alors
 * définissez une propriété ``schema`` dans vos models et retournez
 * simplement ``$Model->schema`` ici à la place.
 */
    public function describe(Model $Model) {
        return $this->_schema;
    }

/**
 * calculate() est pour determiner la façon dont nous allons compter
 * les enregistrements et est requis pour faire fonctionner ``update()``
 * et ``delete()``.
 *
 * Nous ne comptons pas les enregistrements ici mais retournons une chaîne
 * à passer à
 * ``read()`` qui va effectivement faire le comptage. La façon la plus
 * facile est de juste retourner la chaîne 'COUNT' et de la vérifier
 * dans ``read()`` où ``$data['fields'] === 'COUNT'``.
 */
    public function calculate(Model $model, $func, $params = array()) {
        return 'COUNT';
    }

/**
 * Implémente le R dans CRUD. Appel à ``Model::find()`` se trouve ici.
 */
    public function read(Model $model, $queryData = array(), $recursive = null) {
        /**
         * Ici nous faisons réellement le comptage comme demandé par notre
         * méthode calculate() ci-dessus. Nous pouvons soit vérifier la
         * source du dépôt, soit une autre façon pour récupérer le compte
         * de l\'enregistrement. Ici nous allons simplement retourner 1
         * ainsi ``update()`` et ``delete()`` vont estimer que l\'enregistrment
         * existe.
         */
        if ($queryData['fields'] === 'COUNT') {
            return array(array(array('count' => 1)));
        }
        /**
         * Maintenant nous récupèrons, décodons et retournons les données du dépôt.
         */
        $queryData['conditions']['apiKey'] = $this->config['apiKey'];
        $json = $this->Http->get('http://example.com/api/list.json', $queryData['conditions
        $res = json_decode($json, true);
        if (is_null($res)) {
            $error = json_last_error();
```

```
            throw new CakeException($error);
        }
        return array($Model->alias => $res);
    }

/**
 * Implémente le C dans CRUD. Appel à ``Model::save()`` sans $model->id
 * défini se trouve ici.
 */
    public function create(Model $model, $fields = null, $values = null) {
        $data = array_combine($fields, $values);
        $data['apiKey'] = $this->config['apiKey'];
        $json = $this->Http->post('http://example.com/api/set.json', $data);
        $res = json_decode($json, true);
        if (is_null($res)) {
            $error = json_last_error();
            throw new CakeException($error);
        }
        return true;
    }

/**
 * Implémente le U dans CRUD. Appel à ``Model::save()`` avec $Model->id
 * défini se trouve ici. Selon la source du dépôt, vous pouvez just appeler
 * ``$this->create()``.
 */
    public function update(Model $model, $fields = null, $values = null, $conditions = nul
        return $this->create($model, $fields, $values);
    }

/**
 * Implémente le D de CRUD. Appel à ``Model::delete()`` se trouve ici.
 */
    public function delete(Model $model, $id = null) {
        $json = $this->Http->get('http://example.com/api/remove.json', array(
            'id' => $id[$model->alias . '.id'],
            'apiKey' => $this->config['apiKey'],
        ));
        $res = json_decode($json, true);
        if (is_null($res)) {
            $error = json_last_error();
            throw new CakeException($error);
        }
        return true;
    }

}
```

Nous pouvons à présent configurer la source de données dans notre fichier `app/Config/database.php` en y ajoutant quelque chose comme ceci :

```
public $faraway = array(
    'datasource' => 'FarAwaySource',
```

```
    'apiKey'      => '1234abcd',
);
```

Et ensuite utiliser la configuration de notre source de données dans nos models comme ceci :

```
class MyModel extends AppModel {
    public $useDbConfig = 'faraway';
}
```

Nous pouvons à présent récupérer les données depuis notre source distante en utilisant les méthodes familières dans notre model :

```
// Récupère tous les messages de 'Some Person'
$messages = $this->MyModel->find('all', array(
    'conditions' => array('name' => 'Some Person'),
));
```

Astuce : L'utilisation d'autres types de find que `'all'` peut avoir des résultats inattendus si le résultat de votre méthode `read` n'est pas un tableau indexé numériquement.

De la même façon, nous pouvons sauvegarder un nouveau message :

```
$this->MyModel->save(array(
    'name' => 'Some Person',
    'message' => 'New Message',
));
```

Mettre à jour le précédent message :

```
$this->MyModel->id = 42;
$this->MyModel->save(array(
    'message' => 'Updated message',
));
```

Et supprimer le message :

```
$this->MyModel->delete(42);
```

Plugin de source de données

Vous pouvez également empaqueter vos sources de données dans des plugins.

Placez simplement votre fichier de source de données à l'intérieur de `Plugin/[YourPlugin]/Model/Datasource/[YourSource].php` et faites y référence en utilisant la syntaxe pour les plugins :

```
public $faraway = array(
    'datasource' => 'MyPlugin.FarAwaySource',
    'apiKey'      => 'abcd1234',
);
```

Attributs de Model

Les attributs de Model vous permettent de configurer les propriétés qui peuvent surcharger le behavior du model par défaut.

Pour une liste complète d'attributs du model et ses descriptions, visitez l'API de CakePHP. Allez voir http ://api20.cakephp.org/class/model.

useDbConfig

La propriété `useDbConfig` est une chaîne de caractère qui spécifie le nom de la connection à la base de données à utiliser pour lier votre classe model à la table de la base de données liée. Vous pouvez la configurer pour n'importe quelles connexions de base de données définies dans votre fichier de configuration database. Le fichier de configuration database est placé dans /app/Config/database.php.

La propriété `useDbConfig` est par défaut à la connection à la base de données 'default'.

Exemple d'utilisation :

```
class Exemple extends AppModel {
    public $useDbConfig = 'alternate';
}
```

useTable

La propriété `useTable` spécifie le nom de la table de la base de données. Par défaut, le model utilise le nom de classe du model en minuscule, au pluriel. Configurer cette attribut du nom d'une table alternative ou définissez le à `false` si vous souhaitez que le model utilise aucune table de la base de données.

Exemple d'utilisation :

```
class Exemple extends AppModel {
    public $useTable = false; // Ce model n'utilise pas une table de la base de données
}
```

Alternative :

```
class Exemple extends AppModel {
    public $useTable = 'exmp'; // Ce model utilise une table 'exmp' de la base de données
}
```

tablePrefix

Le nom du préfixe de la table utilisé pour le model. Le préfixe de la table est initialement configuré dans le fichier de connection à la base de données dans /app/Config/database.php. Par défaut il n'y a pas de prefix. Vous pouvez écraser la valeur par défaut en configurant l'attribut `tablePrefix` dans le model.

Exemple d'utilisation :

```
class Example extends AppModel {
    public $tablePrefix = 'alternate_'; // va regarder 'alternate_examples'
}
```

primaryKey

Chaque table a normalement une clé primaire, `id`. Vous pouvez changer le nom du champ que le model utlilise en clé primaire. Ceci est courant quand on configure CakePHP pour utiliser une table d'une base de données existante.

Exemple d'utilisation :

```
class Example extends AppModel {
    public $primaryKey = 'example_id'; // example_id est le nom du champ dans la base de d
}
```

displayField

L'attribut `displayField` spécifie quel champ de la base de données doit être utilisé comme label pour un enregistrement. Le label est utilisé dans le scaffolding et dans les appels avec `find('list')`. Le model va utiliser `name` ou `title`, par défaut.

Par exemple, pour utiliser le champ `username` :

```
class User extends AppModel {
    public $displayField = 'username';
}
```

Les noms de champ multiple ne peuvent pas être combinés en un champ unique d'affichage. Par exemple, vous ne pouvez pas spécifier `array('first_name', 'last_name')` en champ à afficher. A la place, créez un champ virtuel avec l'attribut de Model virtualFields

recursive

La propriété recursive définit la profondeur à laquelle CakePHP doit aller attraper les models de données associés via les méthodes `find()`, `findAll()` et `read()`.

Imaginez que votre application dispose de Groups qui appartiennent à un domain et ont plusieurs (many) Users qui à leur tour ont plusieurs (many) Articles. vous pouvez définir $recursive à différentes valeurs basées sur la quantité de données que vous souhaitez retourner à partir d'un appel $this->Group->find() :
– -1 Cake récupère seulement les données de Group, pas de jointures.
– 0 Cake récupère les données de Group et leur domain.
– 1 Cake récupère Group, son domaine et ses Users associés.
– 2 Cake récupère un Group, son domain, ses utilisateurs associés, et les Articles associés des Users.
Ne le définissez pas à plus que vous n'avez besoin. Faire que CakePHP récupère des données dont vous n'aurez pas besoin va ralentir votre application inutilement. Notez aussi que par défaut le niveau de recursive est 1.

Note : Si vous voulez combiner $recursive avec la fonctionnalité `fields`, vous devrez ajouter les colonnes contenant les clés étrangères nécéssaires au tableau `fields` manuellement. Dans l'exemple ci-dessus, ceci pourrait signifier d'ajouter `domain_id`.

Astuce : Le niveau de recursive recommandé pour votre application devrait être -1. Cela évite de récupérer des données liés dans les cas où ce n'est pas nécéssaire ou même non souhaité. C'est le plus souvent le cas pour la plupart de vos appels find(). Augmenter le seulement quand cela est souhaité ou utilisez le behavior Containable.

Vous pouvez réaliser cela en l'ajoutant à AppModel :

```
public $recursive = -1;
```

---

order

L'ordre par défaut des données pour toute opération de type find. Les valeurs possibles incluent :

```
$order = "field"
$order = "Model.field";
$order = "Model.field asc";
$order = "Model.field ASC";
$order = "Model.field DESC";
$order = array("Model.field" => "asc", "Model.field2" => "DESC");
```

data

Le contenu pour les données attrapées pour le model. Alors que les données retournées d'une classe de model sont normalement utilisées à partir d'un appel de find(), vous pourriez avoir besoin d'accéder aux informations stockées dans $data à l'intérieur des callbacks du model.

_schema

Contient les metadata décrivant les champs de la table de la base de données du model. Chaque champ est décrit par :
- name
- type (integer, string, datetime, etc.)
- null
- default value
- length

Exemple d'utilisation :

```
public $_schema = array(
    'first_name' => array(
        'type' => 'string',
        'length' => 30
```

```
    ),
    'last_name' => array(
        'type' => 'string',
        'length' => 30
    ),
    'email' => array(
        'type' => 'string',
        'length' => 30
    ),
    'message' => array('type' => 'text')
);
```

validate

Cet attribut maintient les règles qui permettent au model de faire des décisions de validation de données avant la sauvegarde. Les clés nommées selon les champs maintient les valeurs regex autorisant le model à essayer de faire des correspondances.

Note : Il n'est pas nécéssaire d'appeler validate() avant save() puisque save() va automatiquement valider vos données avant d'effectivement les sauvegarder.

Pour plus d'informations sur la validation, regardez la section suivante *Validation des Donnnées* du manuel.

virtualFields

Tableau de champs virtuels que le model a. Les champs virtuels sont des alias des expressions SQL. Les champs ajoutés à cette propriété vont être lus comme d'autres champs dans un model mais ne seront pas sauvegardables.

Exemple d'utilisation pour MySQL :

```
public $virtualFields = array(
    'name' => "CONCAT(User.first_name, ' ', User.last_name)"
);
```

Dans les opérations ultérieures de find, vos résultats de User contiendraient une clé `name` avec le résultat de la concaténation. Il n'est pas conseillé de créer des champs virtuels avec les mêmes noms comme colonnes dans la base de données, ceci peut causer des erreurs de SQL.

Pour plus d'informations sur la propriété `virtualFields`, son usage propre, ainsi que des limitations, regardez *Champs virtuels*.

name

Nom du model. Si vous ne le spécifiez pas dans votre fichier model, il sera défini automatiquement selon le nom de la classe par le constructeur.

Exemple d'utilisation :

```
class Exemple extends AppModel {
    public $name = 'Exemple';
}
```

cacheQueries

Si défini à true, les données récupérées par le model pendant une requête unique est mis en cache. Cette mise en cache est seulement en mémoire, et dure seulement le temps de la requête. Toute requête dupliquée pour les mêmes données va être gérée par le cache.

Méthodes et Propriétés supplémentaires

Bien que les fonctions de model de CakePHP devraient vous emmener là où vous souhaitez aller, n'oubliez pas que les classes de models ne sont rien de plus que cela : des classes qui vous permettent d'écrire vos propres méthodes ou de définir vos propres propriétés.

N'importe quelle opération qui prend en charge la sauvegarde ou la restitution de données est mieux située dans vos classes de model. Ce concept est souvent appelé fat model ("model gras").

```
class Exemple extends AppModel {
    public function getRecent() {
        $conditions = array(
            'created BETWEEN (curdate() - interval 7 day) and (curdate() - interval 0 day)'
        );
        return $this->find('all', compact('conditions'));
    }
}
```

Cette méthode `getRecent()` peut maintenant être utilisée dans le controller.

```
$recent = $this->Exemple->getRecent();
```

`Model::associations()`

Obtenir les associations :

```
$result = $this->Exemple->associations();
// $result équivaut à array('belongsTo', 'hasOne', 'hasMany', 'hasAndBelongsToMany')
```

`Model::buildQuery(string $type = 'first', array $query = array())`

Construit la requête tableau qui est utilisée par la source de données pour générer la requête pour récupérer les données.

Model::deconstruct(string $field, mixed $data)

Déconstruit un type de données complexe (tableau ou objet) dans une valeur de champ unique.

Model::escapeField(string $field = null, string $alias = null)

Echappe le nom du champ et ajoute le nom du model. L'echappement est fait en fonction des règles du driver de la base de données courante.

Model::exists($id)

Retourne true si l'enregistrement avec un ID particulier existe.

Si l'ID n'est pas fourni, elle appelle `Model::getID()` pour obtenir l'ID de l'enregistrement courant pour vérifier, et execute ensuite un `Model::find('count')` sur la source de données actuellement configurée pour vérifier l'existence de l'enregistrement dans un stockage persistant.

Note : Le Paramètre $id a été ajouté dans 2.1. Avant cela, elle ne prenait aucun paramètre.

```
$this->Exemple->id = 9;
if ($this->Exemple->exists()) {
    // ...
}

$exists = $this->Foo->exists(2);
```

Model::getAffectedRows()

Retourne le nombre de lignes affectées par la dernière requête.

Model::getAssociated(string $type = null)

Récupère tous les models avec lesquels ce model est associé.

Model::getColumnType(string $column)

Retourne le type de colonne d'une colonne du model.

Model::getColumnTypes()

Retourne un tableau associatif des noms de champs et des types de colonnes.

Model::getID(integer $list = 0)

Retourne l'ID de l'enregistrement courant.

Model::getInsertID()

Retourne l'ID du dernier enregistrement que ce model insère.

Model::getLastInsertID()

Alias pour `getInsertID()`.

Champs virtuels

Les champs virtuels vous permettent de créer des expressions SQL arbitraires et de les assigner à des champs dans un Model. Ces champs ne peuvent pas être sauvegardés, mais seront traités comme les autres champs du model pour les opérations de lecture. Ils seront indexés sous la clé du model à travers les autres champs du model.

Créer des champs virtuels

Créer des champs virtuels est facile. Dans chaque model, vous pouvez définir une propriété `$virtualFields` qui contient un tableau de champ => expressions. Un exemple d'une définition de champ virtuel en utilisant MySQL serait :

```
public $virtualFields = array(
    'nom' => 'CONCAT(User.prenom, " ", User.nom_famille)'
);
```

et avec PostgreSQL :

```
public $virtualFields = array(
    'nom' => 'User.prenom || \' \' || User.nom_famille'
);
```

Par conséquent, avec les opérations find, les résultats de l'User contiendraient une clé `nom` avec le résultat de la concaténation. Il n'est pas conseillé de créer des champs virtuels avec les mêmes noms que les colonnes sur la base de données, ce qui peut provoquer des erreurs SQL.

Il n'est pas toujours utile d'avoir **User.prenom** complètement qualifié. Si vous ne suivez pas la convention (ex : vous avez des relations multiples avec d'autres tables) cela entrainerait une erreur. Dans ce cas, il est parfois préferable de juste utiliser `prenom || \'\' || nom` sans le nom du Model.

Utiliser les champs virtuels

Créer des champs virtuels est simple et facile, interagir avec les champs virtuels peut être fait à travers quelques différentes méthodes.

Model : :hasField()

Model : :hasField() retournera true si le model a un champ concret passé en premier paramètre. En définissant le second paramètre de *hasField()* à true, Les champs virtuels seront aussi vérifiés quand on vérifiera si le model a un champ. En utilisant le champ exemple ci-dessus :

```
$this->User->hasField('nom'); // Retournera false, puisqu'il n'y a pas de champ concret app
$this->User->hasField('nom', true); // Retournera true puisqu'il n'y a pas de champ virtuel
```

Model : :isVirtualField()

Cette méthode peut être utilisée pour vérifier si un champ/colonne est un champ virtuel ou champ concret. Retournera true si la colonne est virtuelle :

```
$this->User->isVirtualField('nom'); //true
$this->User->isVirtualField('prenom'); //false
```

Model : :getVirtualField()

Cette méthode peut être utilisée pour accéder aux expressions SQL qui contiennent un champ virtuel. Si aucun argument n'est fourni, il retournera tout champ virtuel dans un Model :

```
$this->User->getVirtualField('nom'); //retoune 'CONCAT(User.prenom, ' ', User.nom_famille)
```

Model : :find() and virtual fields

Comme écrit précédemment, `Model::find()` traitera les champs virtuels un peu comme tout autre champ dans un model. La valeur du champ virtuel sera placé sous la clé du model dans l'ensemble de résultats :

```
$results = $this->User->find('first');

// les résultats contiennent le tableau suivant
array(
    'User' => array(
        'prenom' => 'Mark',
        'nom_famille' => 'Story',
        'nom' => 'Mark Story',
        //plus de champs.
    )
);
```

Pagination et champs virtuels

Depuis que les champs virtuels se comportent un peu plus comme des champs réguliers quand on fait des find, `Controller::paginate()` sera aussi capable de trier selon les champs virtuels.

Champs virtuels et alias de models

Quand on utilise les champsVirtuels et les models avec des alias qui ne sont pas les mêmes que leur nom, on peut se retrouver avec des problèmes comme des champsVirtuels qui ne se mettent pas à jour pour refléter l'alias lié. Si vous utilisez les champsVirtuels dans les models qui ont plus d'un alias, il est mieux de définir les champsVirtuels dans le constructeur de votre model :

```
public function __construct($id = false, $table = null, $ds = null) {
    parent::__construct($id, $table, $ds);
    $this->virtualFields['nom'] = sprintf('CONCAT(%s.prenom, " ", %s.nom_famille)', $this->
}
```

Cel permet à vos champsVirtuels de travailler pour n'importe quel alias que vous donnez à un model.

Champs virtuels dans les requêtes SQL

Utiliser les fonctions dans les requêtes SQL directes assureront que les données seront retournées dans le même tableau que les données du model. Par exemple comme ceci :

```
$this->Timelog->query("SELECT project_id, SUM(id) as TotalHours FROM timelogs AS Timelog GR
```

retourne quelque chose comme ceci :

```
Array
(
    [0] => Array
        (
            [Timelog] => Array
                (
                    [project_id] => 1234
                )
             [0] => Array
                 (
                     [TotalHours] => 25.5
                 )
        )
 )
```

Si nous voulons grouper les TotalHours dans notre tableau de TimeLog, nous devrions spécifier un champ virtuel pour notre colonne aggregée. Nous pouvons ajouter ce nouveau champ virtuel au vol plutôt que de le déclarer de façon permanente dans le model. Nous fournirons une valeur par défaut à `0` au cas où d'autres requêtes attendent d'utiliser ce champ virtuel. Si cela arrive, `0` serait retourné dans la colonne TotalHours :

```
$this->Timelog->virtualFields['TotalHours'] = 0;
```

En plus d'ajouter le champ virtuel, nous avons aussi besoin de faire un alias de notre colonne en utilisant la forme `MonModel__MonChamp` comme ceci :

```
$this->Timelog->query("SELECT project_id, SUM(id) as Timelog__TotalHours FROM timelogs AS T
```

Lancer la requête de nouveau après avoir specifié le champ virtuel résulterait en un groupement plus propre des valeurs :

```
Array
(
    [0] => Array
        (
            [Timelog] => Array
                (
                    [project_id] => 1234
                    [TotalHours] => 25.5
                )
        )
)
```

Limitations des champs virtuels

L'implémentation de `virtualFields` a quelques limitations. Premièrement, vous ne pouvez pas utiliser `virtualFields` sur les models associés pour les conditions, les order, ou les tableaux de champs. Faire ainsi résulte généralement en une erreur SQL puisque les champs ne sont pas remplacés par l'ORM. Cela est du à la difficulté d'estimer la profondeur à laquelle un model associé peut être trouvé.

Une solution de contournement pour ce problème commun de mise en œuvre consiste à copier `virtualFields` d'un model à l'autre lors de l'exécution, lorsque vous avez besoin d'y accéder

```
$this->virtualFields['nom'] = $this->Author->virtualFields['nom'];
```

or :

```
$this->virtualFields += $this->Author->virtualFields;
```

Transactions

Pour effectuer une transaction, les tables d'un model doivent être d'un type qui supporte les transactions.

Toutes les méthodes de transaction doivent être effectuées sur un objet de Source de Données. Pour obtenir le model de Source de Données à partir du model, utilisez :

```
$dataSource = $this->getDataSource();
```

Vous pouvez utiliser la source de données pour commencer, committer, ou faire des roll back des transactions.

```
$dataSource->begin();

// Effectue certaine tâche

if (/*all's well*/) {
    $dataSource->commit();
} else {
```

```
    $dataSource->rollback();
}
```

Les transactions imbriquées

Il est possible de commencer une transaction plusieurs fois en utilisant la méthode `Datasource::begin()`. La transaction va seulement finir quand le nombre de correspondance de *commit* et *rollback* avec ceux du début.

```
$dataSource->begin();
// Execute quelques tâches
$dataSource->begin();
// Quelques tâches en plus
if (/*la dernière tâche ok*/) {
    $dataSource->commit();
} else {
    $dataSource->rollback();
    // Changer quelque chose dans la tâche principale
}
$dataSource->commit();
```

Cela va réaliser une réelle transaction imbriquée si votre base de données le supporte et qu'elle est activée dans la source de données. Les méthodes vont toujours retourner true en mode transaction et quand l'imbrication n'est pas supportée ou désactivée.

Si vous voulez utiliser plusieurs démarrages mais ne pas utiliser la transaction imbriquée à partir de la base de données, désactivez-la en utilisant `$dataSource->useNestedTransactions = false;`. Elle ne va utiliser que la transaction globale.

La transaction réelle imbriquée est désactivée par défaut. Activez-la en utilisant `$dataSource->useNestedTransactions = true;`.

CHAPITRE 7

Librairies du Coeur

CakePHP est fourni avec une pléthore de fonctions et de classes intégrées. Ces classes et fonctions tentent de couvrir certaines des fonctionnalités les plus communes requises dans les applications web.

Usage Général

Des librairies à usage général sont disponibles et réutilisées dans plusieurs endroits de CakePHP.

Usage Général

Constantes globales et Fonctions

Alors que la plupart de vos activités quotidiennes avec CakePHP sera d'initialiser des classes du noyau, CakePHP dispose d'un certain nombre de fonctions globales de confort qui peuvent arriver à point nommé. La plupart de ses fonctions sont à utiliser avec les classes cakePHP (classes de chargement ou de composant), mais beaucoup d'autres rendent le travail avec les tableaux ou les chaînes de caractères un peu plus simple.

Nous allons aussi couvrir une partie des constantes disponibles dans les applications CakePHP. L'utilisation des constantes disponibles vous aidera à faire des mises à jour plus lisses, mais sont aussi des moyens pratiques pour pointer certains fichiers ou répertoires dans vos applications CakePHP.

Fonctions Globales

Voici les fonctions disponibles dans le monde CakePHP. La plupart sont juste des emballages pratiques pour d'autres fonctionnalités CakePHP, comme le débogage et la traduction de contenu.

__ (*string $string_id*[, *$formatArgs*])

Cette fonction gère la localisation dans les applications CakePHP. `$string_id` identifie l'ID de la traduction. Les chaînes utilisées pour la traduction sont traitées comme chaîne formatées

pour `sprintf()`. Vous pouvez fournir des arguments supplémentaires pour remplacer les espaces réservés dans votre chaîne :

```
__('You have %s unread messages', $number);
```

Note : Regardez la section *Internationalisation & Localisation* pour plus d'information.

__c (*string $msg*, *integer $category*, *mixed $args = null*)

Notez que la catégorie doit être spécifiée avec une valeur numérique, au lieu d'un nom de constante. Les valeurs sont :
–0 - LC_ALL
–1 - LC_COLLATE
–2 - LC_CTYPE
–3 - LC_MONETARY
–4 - LC_NUMERIC
–5 - LC_TIME
–6 - LC_MESSAGES

__d (*string $domain*, *string $msg*, *mixed $args = null*)

Vous permet de remplacer le domaine courant lors de la recherche d'un message.

Utile pour internationaliser un plugin : `echo __d('PluginName', 'This is my plugin');`

__dc (*string $domain*, *string $msg*, *integer $category*, *mixed $args = null*)

Vous permet de remplacer le domaine courant pour la recherche d'un message. Permet également de spécifier une catégorie.

Notez que la catégorie doit être spécifiée avec une valeur numérique, au lieu du nom de la constante. Les valeurs sont :
–0 - LC_ALL
–1 - LC_COLLATE
–2 - LC_CTYPE
–3 - LC_MONETARY
–4 - LC_NUMERIC
–5 - LC_TIME
–6 - LC_MESSAGES

__dcn (*string $domain*, *string $singular*, *string $plural*, *integer $count*, *integer $category*, *mixed $args = null*)

Vous permet de remplacer le domaine courant pour la recherche simple au pluriel d'un message. Cela permet également de spécifier une catégorie. Retourne la forme correcte d'un message identifié par $singular et $plural pour le compteur $count depuis le domaine $domain. Certaines langues ont plus d'une forme de pluriel dépendant du compteur.

Notez que la catégorie doit être spécifiée avec des valeurs numériques, au lieu des noms de constantes. Les valeurs sont :
–0 - LC_ALL
–1 - LC_COLLATE
–2 - LC_CTYPE
–3 - LC_MONETARY
–4 - LC_NUMERIC

–5 - LC_TIME
–6 - LC_MESSAGES

__dn (*string $domain*, *string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)
Vous permet de redéfinir le domaine courant pour une recherche simple au pluriel d'un message. Retourne la forme pluriel correcte d'un message identifié par $singular et $plural pour le compteur $count depuis le domaine $domain.

__n (*string $singular*, *string $plural*, *integer $count*, *mixed $args = null*)
Retourne la forme correcte d'un message identifié par $singular et $plural pour le compteur $count. Certaines langues ont plus d'une forme de pluriel dépendant du compteur

am (*array $one*, *$two*, *$three...*)
Fusionne tous les tableaux passés en paramètre et retourne le tableau fusionné.

config ()
Peut être utilisé pour charger des fichiers depuis le dossier config de votre application via include_once. La fonction vérifie l'existence du fichier avant de l'inclure et retourne un booléen. Prend un nombre optionnel d'arguments.

Exemple : `config('un_fichier', 'maconfig');`

convertSlash (*string $string*)
Convertit les slashes en underscores et supprime le premier et le dernier underscores dans une chaîne. Retourne la chaîne convertie.

debug (*mixed $var*, *boolean $showHtml = null*, *$showFrom = true*)
Si le niveau de DEBUG de l'application est différent de zéro, $var est affiché. Si `$showHTML` est true (vrai) ou laissé null, la donnée est formatée pour être visualisée facilement dans un navigateur.

Si `$showFrom` n'est pas défini à false, debug retournera en sortie la ligne depuis laquelle il a été appelé. Voir aussi *Debugger*

env (*string $key*)
Récupère une variable d'environnement depuis les sources disponibles. Utilisé en secours si $_SERVER ou $_ENV sont désactivés.

Cette fonction émule également PHP_SELF et DOCUMENT_ROOT sur les serveurs ne les supportant pas. En fait, c'est une bonne idée de toujours utiliser `env()` plutôt que `$_SERVER` ou ''getenv()' (notamment si vous prévoyez de distribuer le code), puisque c'est un wrapper d'émulation totale.

fileExistsInPath (*string $file*)
Vérifie que le fichier donné est dans le include_path PHP actuel. Renvoie une valeur booléenne.

h (*string $text*, *boolean $double = true*, *string $charset = null*)
Raccourci pratique pour `htmlspecialchars()`.

LogError (*string $message*)
Raccourci pour : `Log::write()`.

pluginSplit (*string $name*, *boolean $dotAppend = false*, *string $plugin = null*)
Divise le nom d'un plugin en notation par point en plugin et classname (nom de classe). Si $name de contient pas de point, alors l'index 0 sera null.

Communément utilisé comme ceci `list($plugin, $name) = pluginSplit('Users.User');`

pr (*mixed $var*)
Raccourci pratique pour `print_r()`, avec un ajout de balises <pre> autour de la sortie.

sortByKey (*array &$array*, *string $sortby*, *string $order = 'asc'*, *integer $type = SORT_NUMERIC*)
Tris de $array par la clé $sortby.

stripslashes_deep (*array $value*)
Enlève récursivement les slashes de la `$valeur` passée. Renvoie le tableau modifié.

Définitions des constantes du noyau

La plupart des constantes suivantes font référence aux chemins dans votre application.

constant APP
Chemin du répertoire de l'application.

constant APP_DIR
La même chose que `app` ou le nom du répertoire de votre application.

constant APPLIBS
Le chemin du répertoire Lib de votre application.

constant CACHE
Chemin vers le répertoire de cache. il peut être partagé entre les hôtes dans une configuration multi-serveurs.

constant CAKE
Chemin vers le répertoire de CAKE.

constant CAKE_CORE_INCLUDE_PATH
Chemin vers la racine du répertoire lib.

constant CORE_PATH
Chemin vers le répertoire racine avec un slash à la fin.

constant CSS
Chemin vers le répertoire CSS publique.

constant CSS_URL
Chemin web vers le répertoire CSS.

constant DS
Raccourci pour la constante PHP DIRECTORY_SEPARATOR, qui est égale à / pour Linux et \ pour Windows.

constant FULL_BASE_URL
Préfix url complet. Comme `https://example.com` Obsolète depuis la version 2.4 : Cette constante est dépréciée, vous devriez utiliser `Router::baseUrl()` à la place.

constant IMAGES
Chemin vers le répertoire images publique.

constant IMAGES_URL
Chemin web vers le répertoire image publique.

constant JS
Chemin vers le répertoire Javascript publique.

constant JS_URL
Chemin web vers le répertoire Javascript publique.

constant LOGS
Chemin du répertoire des logs.

constant ROOT
Chemin vers le répertoire racine.

constant TESTS
Chemin vers le répertoire de test.

constant TMP
Chemin vers le répertoire des fichiers temporaires.

constant VENDORS
Chemin vers le répertoire vendors.

constant WEBROOT_DIR
La même chose que `webroot` ou le nom du répertoire webroot.

constant WWW_ROOT
Chemin d'accès complet vers la racine web (webroot).

Définition de Constantes de Temps

constant TIME_START
timestamp Unix en microseconde au format float du démarrage de l'application.

constant SECOND
Égale à 1

constant MINUTE
Égale à 60

constant HOUR
Égale à 3600

constant DAY
Égale à 86400

constant WEEK
Égale à 604800

constant MONTH
Égale à 2592000

constant YEAR
Égale à 31536000

Classe App

class App

La classe App est responsable de la gestion des chemins, la classe de localisation et la classe de chargement. Assurez-vous que vous suivez les *Conventions des Fichiers et des Noms de Classe*.

Packages

CakePHP est organisé autour de l'idée de packages, chaque classe appartient à un package ou dossier où d'autres classes se trouvent. Vous pouvez configurer chaque localisation de package dans votre application en utilisant `App::build('APackage/SubPackage', $paths)` pour informer le framework où chaque classe doit être chargée. Presque toute classe dans le framework CakePHP peut être échangée avec une des votres compatible. Si vous souhaitez utiliser votre propre classe à la place des classes que le framework fournit, ajoutez seulement la classe à vos dossiers libs émulant la localisation du répertoire à partir duquel CakePHP s'attend à le trouver.

Par exemple, si vous voulez utiliser votre propre classe HttpSocket, mettez la sous :

```
app/Lib/Network/Http/HttpSocket.php
```

Une fois ceci fait, cette App va charger votre fichier à la place du fichier de l'intérieur de CakePHP.

Chargement des classes

static `App::uses` (*string $class*, *string $package*)

Type retourné void

Les classes sont chargées toutes seules dans CakePHP, cependant avant que l'autoloader puisse trouver vos classes que vous avez besoin de dire à App, où il peut trouver les fichiers. En disant à App dans quel package une classe peut être trouvée, il peut bien situer le fichier et le charger la première fois qu'une classe est utilisée.

Quelques exemples pour les types courants de classes sont :

Controller `App::uses('PostsController', 'Controller');`

Component `App::uses('AuthComponent', 'Controller/Component');`

Model `App::uses('MyModel', 'Model');`

Behaviors `App::uses('TreeBehavior', 'Model/Behavior');`

Views `App::uses('ThemeView', 'View');`

Helpers `App::uses('HtmlHelper', 'View/Helper');`

Libs `App::uses('PaymentProcessor', 'Lib');`

Vendors `App::uses('Textile', 'Vendor');`

Utility `App::uses('String', 'Utility');`

Donc au fond, le deuxième paramètre devrait simplement correspondre au chemin du dossier de la classe de fichier dans core ou app.

Note : Charger des vendors signifie généralement que vous chargez des packages qui ne suivent pas les conventions. Pour la plupart des packages vendor, l'utilisation de `App::import()` est recommandée.

Chargement des fichiers à partir des plugins Le chargement des classes dans les plugins fonctionnnent un peu de la même façon que le chargement des classes app et des classes du coeur sauf que vous devez spécifier le plugin à partir du quel vous chargez :

```
// Charge la classe Comment dans app/Plugin/PluginName/Model/Comment.php
App::uses('Comment', 'PluginName.Model');

// Charge la classe class CommentComponent dans app/Plugin/PluginName/Controller/Component/
App::uses('CommentComponent', 'PluginName.Controller/Component');
```

Trouver des chemins vers les packages en utilisant App : :path()

static `App::path` (*string $package*, *string $plugin = null*)

Type retourné array

Utilisé pour lire les informations sur les chemins stockés :

```
// retourne les chemins de model dans votre application
App::path('Model');
```

Ceci peut être fait pour tous les packages qui font parti de votre application. VOus pouvez aussi récupérer des chemins pour un plugin :

```
// retourne les chemins de component dans DebugKit
App::path('Component', 'DebugKit');
```

static `App::paths` ()

Type retourné array

Récupère tous les chemins chargés actuellement à partir de App. Utile pour inspecter ou stocker tous les chemins que App connait. Pour un chemin vers un package spécifique, utilisez `App::path()`.

static `App::core` (*string $package*)

Type retourné array

Utilisé pour trouver le chemin vers un package à l'intérieur de CakePHP :

```
// Récupère le chemin vers les moteurs de Cache.
App::core('Cache/Engine');
```

static `App::location` (*string $className*)

Type retourné string

Retourne le nom du package d'où une classe a été localisée.

Ajoutez des chemins dans App pour trouver des packages

static `App::build`(*array $paths = array()*, *mixed $mode = App : :PREPEND*)

Type retourné void

Définit chaque localisaiton de package dans le système de fichier. Vous pouvez configurer des chemins de recherche multiples pour chaque package, ceux-ci vont être utilisés pour rechercher les fichiers, un dossier à la fois, dans l'ordre spécifié. Tous les chemins devraient être terminés par un séparateur de répertoire.

Ajouter des chemins de controller supplémentaires pourraient par exemple modifier où CakePHP regarde pour les controllers. Cela vous permet de séparer votre application à travers le système de fichier.

Utilisation :

```
//Va configurer un nouveau chemin de recherche pour le package Model
App::build(array('Model' => array('/a/full/path/to/models/')));

//Va configurer le chemin comme le seule chemin valide pour chercher les models
App::build(array('Model' => array('/path/to/models/')), App::RESET);

//Va configurer les chemins de recherche multiple pour les helpers
App::build(array('View/Helper' => array('/path/to/helpers/', '/another/path/')));
```

Si reset est défini à true, tous les plugins chargés seront oubliés et ils devront être rechargés.

Exemples :

```
App::build(array('controllers' => array('/full/path/to/controllers/')))
//devient
App::build(array('Controller' => array('/full/path/to/Controller/')))

App::build(array('helpers' => array('/full/path/to/views/helpers/')))
//devient
App::build(array('View/Helper' => array('/full/path/to/View/Helper/')))
```

Modifié dans la version 2.0 : `App::build()` ne va plus fusionner les chemins de app avec les chemins du coeur.

Ajoutez de nouveaux packages vers une application `App::build()` peut être utilisé pour ajouter de nouvelles localisations de package. Ceci est utile quand vous voulez ajouter de nouveaux packages de niveaux supérieurs ou, des sous-packages à votre application :

```
App::build(array(
    'Service' => array('%s' . 'Service' . DS)
), App::REGISTER);
```

Le `%s` dans les packages nouvellement enregistrés, sera remplacé par le chemin `APP`. Vous devez inclure un trailing `/` dans les packages enregistrés. Une fois que les packages sont enregistrés, vous pouvez utiliser `App::build()` pour ajouter/préfixer/remettre les chemins comme dans tout autre package. Modifié dans la version 2.1 : Les packages enregistrés a été ajouté dans 2.1

Trouver les objets que CakePHP connaît

static `App::objects` (*string $type*, *mixed $path = null*, *boolean $cache = true*)

Type retourné mixed Retourne un tableau d'objets du type donné ou à false si incorrect

Vous pouvez trouver quels objets App connaît en utilisant `App::objects('Controller')` par exemple pour trouver quels controllers de l'application App connaît.

Exemple d'utilisation :

```
//retourne array('DebugKit', 'Blog', 'User');
App::objects('plugin');

//retourne array('PagesController', 'BlogController');
App::objects('Controller');
```

Vous pouvez aussi chercher seulement dans les objets de plugin en utilisant la syntaxe de plugin avec les points. :

```
// retourne array('MyPluginPost', 'MyPluginComment');
App::objects('MyPlugin.Model');
```

Modifié dans la version 2.0.

1. Retourne `array()` au lieu de false pour les résultats vides ou les types invalides.
2. Ne retourne plus les objets du coeur, `App::objects('core')` retournera `array()`.
3. Retourne le nom de classe complet.

Localiser les plugins

static `App::pluginPath` (*string $plugin*)

Type retourné string

Les Plugins peuvent être localisés aussi avec App. En utilisant `App::pluginPath('DebugKit');` par exemple, vous donnera le chemin complet vers le plugin DebugKit :

```
$path = App::pluginPath('DebugKit');
```

Localiser les thèmes

static `App::themePath` (*string $theme*)

Type retourné string

Les Thèmes peuvent être trouvés `App::themePath('purple');`, vous donnerait le chemin complet vers le thème *purple*.

Inclure les fichiers avec App : :import()

static `App::import` (*mixed $type = null*, *string $name = null*, *mixed $parent = true*, *array $search = array()*, *string $file = null*, *boolean $return = false*)

Type retourné boolean

Au premier coup d'oeil, `App::import` a l'air compliqué, cependant pour la plupart des utilisations, seuls 2 arguments sont nécessaires.

Note : Cette méthode est équivalente à faire un `require` sur le fichier. Il est important de réaliser que la classe doit ensuite être initialisée.

```
// La même chose que require('Controller/UsersController.php');
App::import('Controller', 'Users');

// Nous avons besoin de charger la classe
$Users = new UsersController;

// Si nous voulons que les associations de model, les components, etc
soient chargées
$Users->constructClasses();
```

Toutes les classes qui sont chargées dans le passé utilisant App : :import('Core', $class) devront être chargées en utilisant App : :uses() se référant au bon package. Ce changement a fourni de grands gains de performances au framework. Modifié dans la version 2.0.

–Cette méthode ne regarde plus les classes de façon récursive, elle utilise strictement les valeurs pour les chemins définis dans `App::build()`.

–Elle ne sera pas capable de charger `App::import('Component', 'Component')`, utilisez `App::uses('Component', 'Controller');`.

–Utilisez `App::import('Lib', 'CoreClass');` pour charger les classes du coeur n'est plus possible.

–Importer un fichier non existant, fournir un mauvais type ou un mauvais nom de package, ou des valeurs null pour les paramètres `$name` et `$file` entraînera une valeur de retour à false.

–`App::import('Core', 'CoreClass')` n'est plus supporté, utilisez `App::uses()` à la place et laissez la classe d'autochargement faire le reste.

–Charger des fichiers de Chargement ne regarde pas de façon récursive dans le dossier vendors, il ne convertira plus aussi le fichier avec des underscores comme il le faisait dans le passé.

Surcharger les classes dans CakePHP

Vous pouvez surcharger presque toute classe dans le framework, les exceptions sont les classes `App` et `Configure`. Quelque soit le moment où vous souhaitez effectuer l'écrasement, ajoutez seulement votre classe dans votre dossier app/Lib en imitant la structure interne du framework. Quelques exemples suivants

– Pour écraser la classe `Dispatcher`, créer `app/Lib/Routing/Dispatcher.php`.
– Pour écraser la classe `CakeRoute`, créer `app/Lib/Routing/Route/CakeRoute.php`.
– Pour écraser la classe `Model`, créer `app/Lib/Model/Model.php`.

Quand vous chargez les fichiers remplacés, les fichiers de app/Lib seront chargés à la place des classes intégrées au coeur.

Charger des fichiers Vendor

Vous pouvez utiliser `App::uses()` pour charger des classes provenant des répertoires vendors. Elle suit les mêmes conventions que pour le chargement des autres fichiers :

```
// Charge la classe Geshi dans app/Vendor/Geshi.php
App::uses('Geshi', 'Vendor');
```

Pour charger les classes se trouvant dans des sous-répertoires, vous devrez ajouter ces chemins avec `App::build()` :

```
// Charge la classe ClassInSomePackage dans app/Vendor/SomePackage/ClassInSomePackage.php
App::build(array('Vendor' => array(APP . 'Vendor' . DS . 'SomePackage')));
App::uses('ClassInSomePackage', 'Vendor');
```

Vos fichiers vendor ne suivent peut-être pas les conventions, ont une classe qui diffère du nom de fichier ou ne contiennent pas de classes. Vous pouvez charger ces fichiers en utilisant `App::import()`. Les exemples siuvants montrent comment charger les fichiers de vendor à partir d'un certain nombre de structures de chemin. Ces fichiers vendor pourrraient être localisés dans n'importe quel dossier vendor.

Pour charger **app/Vendor/geshi.php** :

```
App::import('Vendor', 'geshi');
```

Note : Le nom du fichier geshi doit être en minuscule puisque Cake ne le trouvera pas sinon.

Pour charger **app/Vendor/flickr/flickr.php** :

```
App::import('Vendor', 'flickr/flickr');
```

Pour charger **app/Vendor/some.name.php** :

```
App::import('Vendor', 'SomeName', array('file' => 'some.name.php'));
```

Pour charger **app/Vendor/services/well.named.php** :

```
App::import('Vendor', 'WellNamed', array('file' => 'services' . DS . 'well.named.php'));
```

Cela ne ferait pas de différence si vos fichiers vendor étaient à l'intérieur du répertoire /vendors. Cake le trouvera automatiquement.

Pour charger **vendors/vendorName/libFile.php** :

```
App::import('Vendor', 'aUniqueIdentifier', array('file' => 'vendorName' .DS . 'libFile.php'
```

Les Méthodes Init/Load/Shutdown de App

static `App::init()`

Type retourné void

Initialise le cache pour App, enregistre une fonction shutdown (fermeture).

static `App::load(string $className)`

Type retourné boolean

Méthode pour la gestion automatique des classes. Elle cherchera chaque package de classe défini en utilisant `App::uses()` et avec cette information, elle va transformer le nom du package en un chemin complet pour charger la classe. Le nom de fichier pour chaque classe devrait suivre le nom de classe. Par exemple, si une classe est nommée `MyCustomClass` le nom de fichier devrait être `MyCustomClass.php`.

static `App::shutdown()`

Type retourné void

Destructeur de l'Objet. Ecrit le fichier de cache si les changements ont été faits à `$_map`.

Événements système

Introduit dans la version 2.1. La création d'applications maintenables est à la fois une science et un art. Il est connu que la clé pour avoir des codes de bonne qualité est d'avoir un couplage plus lâche et une cohésion plus élevée. La cohésion signifie que toutes les méthodes et propriétés pour une classe sont fortement reliés à la classe elle même et non pas d'essayer de faire le travail que d'autre objets devraient faire, tandis que un couplage plus lâche est la mesure du degré de resserrement des interconnexions d'une classe aux objets externes, et comment cette classe en dépend.

Tandis que la plupart des structures CakePHP et des librairies par défaut vous aideront à atteindre ce but, il y a certains cas où vous avez besoin de communiquer proprement avec les autres parties du système sans avoir à coder en dur ces dépendances, ainsi réduire la cohésion et accroître le couplage de classe. Un motif de conception (design pattern) très réussi dans l'ingénierie software est le modèle obervateur (Observer pattern), où les objets peuvent générer des événements et notifier à des écouteurs (listener) possiblement anonymes des changements d'états internes.

Les écouteurs (listener) dans le modèle observateur (Observer pattern) peuvent s'abonner à de tel événements et choisir d'interagir sur eux, modifier l'état du sujet ou simplement faire des logs. Si vous avez utilisez javascript dans le passé, vous avez la chance d'être déjà familier avec la programmation événementielle.

CakePHP émule plusieurs aspects sur la façon dont les événements sont déclenchés et managés dans des frameworks javascript comme le populaire jQuery, tout en restant fidèle à sa conception orientée objet. Dans cette implémentation, un objet événement est transporté a travers tous les écouteurs qui détiennent l'information et la possibilité d'arrêter la propagation des événements à tout moment. Les écouteurs peuvent s'enregistrer eux-mêmes ou peuvent déléguer cette tâche a d'autres objets et avoir la chance de modifier l'état et l'événement lui-même pour le reste des callbacks.

Interagir avec le gestionnaire d'événement

Imaginons que vous être en train de construire un plugin Caddie, mais vous ne voulez pas vraiment l'encombrer avec une logique d'expédition, expédier un mail à l'utilisateur ou décrémenter les articles depuis le stock, c'est votre souhait de traiter tout cela séparément dans un autre plugin ou dans le code de l'application. Typiquement, quand vous n'utilisez pas directement le modèle observateur(observer pattern) vous voudriez faire cela en attachant des behaviors à la volée à vos modèles, et peut être quelques components aux controllers.

Faire cela représente un challenge la plupart du temps, puisque vous allez devoir aborder du code pour charger de manière externe ces behaviors ou d'attacher des hooks aux controllers de votre plugin. Avant CakePHP 2.1 quelques développeurs choisissaient d'implémenter des événements systèmes génériques pour résoudre ce problème, et certain de ces systèmes étaient offerts comme plugins. Maintenant, vous pouvez bénéficier en standard d'un système d'événements qui vous laissera proprement séparer ce qui concerne vos plugins et le code de l'application avec le gestionnaire d'événements intégré.

Distribution des événements Revenons à notre exemple, nous aurions un model *Order* qui gèrera la logique d'achat, et probablement une méthode *place* pour enregistrer les détails de la commande et faire d'autres logiques :

```
// Cart/Model/Order.php
class Order extends AppModel {

    public function place($order) {
        if ($this->save($order)) {
            $this->Cart->remove($order);
            $this->sendNotificationEmail();
            $this->decrementFromStock();
            $this->updateUserStatistics();
            // ...
            return true;
        }
        return false;
    }
}
```

Bien, cela ne semble pas correct du tout. Un plugin ne devrait pas faire d'hypothèse sur l'envoi d'email, et devrait même ne pas avoir à décrémenter les données d'inventaire, et finalement le suivi statistique n'est pas le meilleur endroit pour le faire. Nous avons donc besoin d'une autre solution, ré-écrivons en utilisant le gestionnaire d'événements.

```
// Cart/Model/Order.php
App::uses('CakeEvent', 'Event');
class Order extends AppModel {

    public function place($order) {
        if ($this->save($order)) {
            $this->Cart->remove($order);
            $this->getEventManager()->dispatch(new CakeEvent('Model.Order.afterPlace', $thi
                'order' => $order
            )));
```

```
            return true;
        }
        return false;
    }
}
```

Cela semble beaucoup plus clair, et nous donne l'opportunité d'introduire des classes d'événement et des méthodes. La première chose que vous devriez noter c'est l'appel à `getEventManager()` qui est une méthode disponible par défaut dans tous les Models, Controller, et les Vues. Cette méthode ne retournera pas la même instance du gestionnaire aux travers des models, et n'est pas partagée entre les controllers et les models, mais elle l'est entre les controllers et les vues, cependant. Nous reverrons plus tard comment surmonter ce détail d'implémentation.

La méthode `getEventManager` retourne une instance de `CakeEventManager`, et pour dispatcher les événements vous utilisez `CakeEventManager::dispatch()` qui reçoit une instance de la classe `CakeEvent`. Disséquons maintenant le processus de dispatching d'un événement :

```
new CakeEvent('Model.Order.afterPlace', $this, array(
    'order' => $order
));
```

`CakeEvent` reçoit 3 arguments dans son constructeur. Le premier est le nom de l'événement, vous devrez essayer de garder ce nom aussi unique que possible, tout en le rendant lisible. Nous vous suggérons les conventions suivantes : *Layer.eventName* pour les événements généraux qui surviennent à un niveau de couche(ex :*Controller.startup*,'View.beforeRender') et *Layer.Class.eventName* pour les événements qui surviennent dans une classe spécifique sur une couche, par exemple *Model.User.afterRegister* ou *Controller.Courses.invalidAccess*.

Le second argument est le sujet *subject*, ce qui signifie l'objet associé à l'événement, habituellement quand c'est la même classe de déclenchement d'événements que lui même, *$this* sera communément utilisé. Bien que `Component` pourrait lui aussi déclencher les événements du controller. La classe du sujet est importante parce que les écouteurs (listeners) auront des accès immédiats aux propriétés des objets et la chance de les inspecter ou de les changer à la volée.

Finalement, le troisième argument est le paramètres d'événement. Ceci peut être n'importe quelle donnée que vous considérez comme étant utile à passer avec laquelle les écouteurs peuvent interagir. Même si cela peut être n'importe quel type d'argument, nous vous recommandons de passer un tableau associatif, pour rendre l'inspection plus facile.

La méthode `CakeEventManager::dispatch()` accepte les objets événements comme arguments et notifie a tous les écouteurs et callbacks le passage de cet objet. Ainsi les écouteurs géreront toute la logique autour de l'événement *afterPlace*, vous pouvez enregistrer l'heure, envoyer des emails, éventuellement mettre à jour les statistiques de l'utilisateur dans des objets séparés et même déléguer cela à des tâches hors-ligne si vous en avez besoin.

Enregistrement des callbacks Comment allons nous enregistrer les callbacks ou les observateurs (observers) dans notre nouvel événement *afterPlace* ? Ceci est sujet à une grande variété d'implémentation, mais elles ont toutes à appeler la méthode `CakeEventManager::attach()` pour enregistrer les nouveaux acteurs. Par souci de simplicité, imaginons que nous savons dans le plugin de quoi les callbacks sont

capables dans le controller, et disons que ce controller est responsable de leur attachement. Le code possible pourrait être cela

```
// Les écouteurs (Listeners) configurés quelque part ailleurs, un fichier de config par ex
Configure::write('Order.afterPlace', array(
    'email-sending' => 'EmailSender::sendBuyEmail',
    'inventory' => array($this->InventoryManager, 'decrement'),
    'logger' => function($event) {
        // Les fonction Anonyme ne sont permises que dans PHP 5.3+
        CakeLog::write('info', 'Un nouvel achat à été placé avec l\'id: ' . $event->subject
    }
));

// Cart/Controller/OrdersController.php
class OrdersController extends AppController {

    public function finish() {
        foreach (Configure::read('Order.afterPlace') as $l) {
            $this->Order->getEventManager()->attach($l, 'Model.Order.afterPlace');
        }
        if ($this->Order->place($this->Cart->items())) {
            // ...
        }
    }
}
```

Ce ne sera pas la façon la plus propre de la faire, ainsi nous pouvons arriver avec nos propres moyens pour attacher des écouteurs au gestionnaire d'événement d'un objet. Cette façon simple de les définir en utilisant la classe *Configure* est destinée a des fins didactiques seulement. Ce petit exemple nous permet de montrer quel type de callbacks peuvent être attachés au gestionnaire (manager). Comme vous vous en doutiez, la méthode *attach* prend n'importe quel type de *callback* PHP valide, c'est une chaîne représentant l'appel à une fonction statique, un tableau ayant une instance de classe et une méthode, une fonction anonyme si vous utilisez PHP 5.3 ,etc... Les callbacks attachés recevront l'objet événement comme premier argument.

`CakeEventManager::attach()` accepte trois arguments. Le plus à gauche est le callback lui même, n'importe quoi que PHP peut traiter comme une fonction appelable. Le second argument est le nom de l'événement, et le callback ne sera envoyé que si l'objet *CakeEvent* envoyé a un nom correspondant. Le dernier argument est un tableau d'options pour configurer la priorité du callback,et les arguments préférentiels à envoyer.

Enregistrer les écouteurs Les écouteurs (Listeners) sont une alternative, et souvent le moyen le plus propre d'enregistrer les callbacks pour un événement. Ceci est fait en implémentant l'interface `CakeEventListener` dans chacune de classes ou vous souhaitez enregistrer des callbacks. Les classes l'implémentant doivent fournir la méthode `implementedEvents()` et retourner un tableau associatif avec tous les noms d'événements que la classe gérera.

Pour en revenir à notre exemple précédent, imaginons que nous avons une classe UserStatistic responsable du calcul d'information utiles et de la compilation de statistiques dans le site global. Ce serait naturel de passer une instance de cette classe comme un callback, au lien d'implémenter une fonction statique personnalisé ou la conversion de n'importe quel autre contournement pour déclencher les méthodes de cette classe. Un écouteur (listener) est créé comme ci-dessous

```
App::uses('CakeEventListener', 'Event');
class UserStatistic implements CakeEventListener {

    public function implementedEvents() {
        return array(
            'Model.Order.afterPlace' => 'updateBuyStatistic',
        );
    }

    public function updateBuyStatistic($event) {
        // Code pour mettre à jour les statistiques
    }
}

// Attache l'objet UserStatistic au gestionnaire d'événement 'Order' (commande)
$statistics = new UserStatistic();
$this->Order->getEventManager()->attach($statistics);
```

Comme vous pouvez le voir dans le code ci-dessus, la fonction *attach* peut manipuler les instances de l'interface *CakeEventListener*. En interne, le gestionnaire d'événement lira le tableau retourné par la méthode *implementedEvents* et relie les callbacks en conséquence.

Établir des priorités Dans certains cas, vous souhaitez exécuter un callback et être sûre qu'il sera exécuté avant, ou après tous les autres callbacks déjà lancés. Par exemple, repensons a notre exemple de statistiques utilisateur. Il serait judicieux de n'exécuter cette méthode que si nous sommes sûrs que l'événement n'a pas été annulé, qu'il n'y a pas d'erreur et que les autres callbacks n'ont pas changés l'état de 'order' lui même. Pour ces raisons vous pouvez utiliser les priorités.

Les priorités sont gérés en utilisant un nombre associé au callback lui même. Plus haut est le nombre, plus tard sera lancée la méthode. Les priorités par défaut des méthodes des callbacks et écouteurs sont définis à '10'. Si vous voulez que votre méthode soit lancée avant, alors l'utilisation de n'importe quelle valeur plus basse que cette valeur par défaut vous aidera à le faire, même en mettant la priorité à *1* ou une valeur négative pourrait fonctionner. D'une autre façon si vous désirez exécuter le callback après les autres, l'usage d'un nombre au dessus de *10* fonctionnera.

Si deux callbacks se trouvent alloués avec le même niveau de priorité, ils seront exécutés avec une règle *FIFO*, la première méthode d'écouteur (listener) attachée est appelée en premier et ainsi de suite. Vous définissez les priorités en utilisant la méthode *attach* pour les callbacks, et les déclarer dans une fonction *implementedEvents* pour les écouteurs d'événements :

```
// Paramétrage des priorités pour un callback
$callback = array($this, 'doSomething');
$this->getEventManager()->attach($callback, 'Model.Order.afterPlace', array('priority' => 2

// Paramétrage des priorité pour un écouteur(listener)
class UserStatistic implements CakeEventListener {
    public function implementedEvents() {
        return array(
            'Model.Order.afterPlace' => array('callable' => 'updateBuyStatistic', 'priorit
        );
```

```
    }
}
```

Comme vous pouvez le voir, la principale différence pour les objets *CakeEventListener* c'est que vous avez à utiliser un tableau pour spécifier les méthodes appelables et les préférences de priorités. La clé appelable *callable* est une entrée de tableau spéciale que le gestionnaire (manager) lira pour savoir quelle fonction dans la classe il devrait appeler.

Obtenir des données d'événements comme paramètres de fonction Certain développeurs pourraient préférer avoir les données d'événements passées comme des paramètres de fonctions au lieu de recevoir l'objet événement. Bien que ce soit un préférence étrange et que l'utilisation d'objet événement est bien plus puissant, ceci a été nécessaire pour fournir une compatibilité ascendante avec le précédent système d'événement et pour offrir aux développeurs chevronnés une alternative pour ce auquel ils sont habitués.

Afin de changer cette option, vous devez ajouter l'option *passParams* au troisième argument de la méthode *attach*, ou le déclarer dans le tableau de retour *implementedEvents* de la même façon qu'avec les priorités :

```
// Paramétrage des priorités pour le callback
$callback = array($this, 'doSomething');
$this->getEventManager()->attach($callback, 'Model.Order.afterPlace', array('passParams' =>

// Paramétrage des priorités pour l'écouteur (listener)
class UserStatistic implements CakeEventListener {
    public function implementedEvents() {
        return array(
            'Model.Order.afterPlace' => array('callable' => 'updateBuyStatistic', 'passPara
        );
    }

    public function updateBuyStatistic($orderData) {
        // ...
    }
}
```

Dans l'exemple ci-dessus la fonction *doSomething* et la méthode *updateBuyStatistic* recevrons *$orderData* au lieu de l'objet *$event*. C'est comme cela parce que dans notre premier exemple nous avons déclenché l'événement *Model.Order.afterPlace* avec quelques données :

```
$this->getEventManager()->dispatch(new CakeEvent('Model.Order.afterPlace', $this, array(
    'order' => $order
)));
```

Note : Les paramètres ne peuvent être passés comme arguments de fonction que si la donnée d'événement est un tableau. N'importe quelle autre type de données sera converti en paramètre de fonction, ne pas utiliser cette option est souvent plus adéquate.

Stopper des événements Il y a des circonstances ou vous aurez besoin de stopper des événements de sorte que l'opération commencée est annulée. Vous voyez un exemple de cela dans les callbacks des models (ex.

beforesave) dans lesquels il est possible de stopper une opération de sauvegarde si le code détecte qu'il ne peut pas aller plus loin.

Afin de stopper les événements vous pouvez soit retourner *false* dans vos callbacks ou appeler la méthode *stopPropagation* sur l'objet événement :

```
public function doSomething($event) {
    // ...
    return false; // stoppe l'événement
}

public function updateBuyStatistic($event) {
    // ...
    $event->stopPropagation();
}
```

Stopper un événement peut avoir deux effets différents. Le premier peut toujours être attendu ; n'importe quel callback après l'événement qui à été stoppé ne sera appelé. Le seconde conséquence est optionnelle et dépend du code qui déclenche l'événement, par exemple, dans votre exemple *afterPlace* cela n'aurait pas de sens d'annuler l'opération tant que les données n'aurons pas toutes été enregistrées et le Caddie vidé. Néanmoins, si nous avons une *beforePlace* arrêtant l'événement cela semble valable.

Pour vérifier qu'un événement à été stoppé, vous appelez la méthode *isStopped()* dans l'objet événement :

```
public function place($order) {
    $event = new CakeEvent('Model.Order.beforePlace', $this, array('order' => $order));
    $this->getEventManager()->dispatch($event);
    if ($event->isStopped()) {
        return false;
    }
    if ($this->Order->save($order)) {
        // ...
    }
    // ...
}
```

Dans l'exemple précédent la vente ne seraient pas enregistrée si l'événement est stoppé durant le processus *beforePlace*.

Récupérer les résultats d'événement Chacune des fois ou un callback retourne une valeur, celle ci est stockée dans la propriété *$result* de l'objet événement. C'est utile dans certains cas où laisser les callbacks modifier les paramètres principaux de processus augmente la possibilité de modifier l'aspect d'exécution des processus. Regardons encore notre exemple *beforePlace* et laissons les callbacks modifier la donnée $order (commande).

Les résultats d'événement peuvent être modifiés soit en utilisant directement la propriété result de l'objet event ou en retournant une valeur dans le callback lui même. :

```
// Un écouteur (listener) de callback
public function doSomething($event) {
    // ...
    $alteredData = $event->data['order']  + $moreData;
```

```
    return $alteredData;
}

// Un autre écouteur (listener) de callback
public function doSomethingElse($event) {
    // ...
    $event->result['order'] = $alteredData;
}

// Utilisation du résultat de l'événement
public function place($order) {
    $event = new CakeEvent('Model.Order.beforePlace', $this, array('order' => $order));
    $this->getEventManager()->dispatch($event);
    if (!empty($event->result['order'])) {
        $order = $event->result['order'];
    }
    if ($this->Order->save($order)) {
        // ...
    }
    // ...
}
```

Comme vous l'avez peut-être aussi remarqué, il est possible de modifier n'importe quelle propriété d'un objet événement et d'être sûr que ces nouvelles données seront passées au prochain callback. Dans la majeur partie des cas, fournir des objets comme donnée événement ou résultat et modifier directement les objets est la meilleur solution puisque la référence est maintenue et que les modifications sont partagées à travers les appels callbacks.

Retirer des callbacks et écouteurs (listeners) Si pour quelque raison que ce soit, vous voulez retirer certains callbacks depuis le gestionnaire d'événement, appelez juste la méthode `CakeEventManager::detach()` en utilisant comme arguments les deux premiers paramètres que vous avez utilisés pour les attacher

```
// Attacher une fonction
$this->getEventManager()->attach(array($this, 'doSomething'), 'My.event');

// Détacher la fonction
$this->getEventManager()->detach(array($this, 'doSomething'), 'My.event');

// Attacher une fonction anonyme (PHP 5.3+ seulement);
$myFunction = function($event) { ... };
$this->getEventManager()->attach($myFunction, 'My.event');

// Détacher la fonction anonyme
$this->getEventManager()->detach($myFunction, 'My.event');

// Attacher un écouteur Cake (CakeEventListener)
$listener = new MyCakeEventLister();
$this->getEventManager()->attach($listener);

// Détacher une simple clé d'événement depuis un écouteur (listener)
```

```
$this->getEventManager()->detach($listener, 'My.event');

// Détacher tous les callbacks implémentés par un écouteur (listener)
$this->getEventManager()->attach($listener);
```

Le gestionnaire d'événement global

Comme noté précédemment, cela pourrait être difficile d'attacher les observateurs à un gestionnaire d'événement particulier dans un objet. il y a certains cas où avoir la possibilité d'attacher des callbacks à un événement est nécessaire sans avoir accès à l'instance objet qui le déclenchera. Aussi, pour empêcher les gens d'implémenter pour chacun d'eux un mécanisme différent pour le chargement des callbacks dans les gestionnaires en fonction de la configuration, CakePHP fournit le concept du gestionnaire d'événement global *global event manager*.

Le gestionnaire global est une instance singleton de la classe `CakeEventManager` qui reçoit tous les événements que n'importe quel gestionnaire d'événement dans l'application dispatches. C'est à la fois flexible et puissant, mais si vous l'utilisez vous devez prendre plus de précautions avec le traitement des événements.

Pour définir le concept une fois de plus, et utiliser notre exemple *beforePlace* souvenons nous que nous utilisions le gestionnaire d'événement local qui est retourné par la fonction *getEventManager*. En interne ce gestionnaire d'événement local dispatche l'événement dans le gestionnaire d'événement global avant de déclencher les callbacks internes attachés. La priorité de chacun des managers est indépendante, le callback global se lancera dans leur propre file d'attente et ainsi les callbacks locaux seront appelés dans l'ordre de priorité respectifs.

Accéder au gestionnaire d'événement global est aussi simple que d'appeler une fonction statique, l'exemple suivant va attacher un événement global à l'événement *beforePlace*

```
// Dans n'importe quel fichier de config ou morceau de code qui s'exécute avant l'événement
App::uses('CakeEventManager', 'Event');
CakeEventManager::instance()->attach($aCallback, 'Model.Order.beforePlace');
```

Comme vous pouvez le voir, nous avons juste changé comment nous avons accès à l'instance du gestionnaire d'événement, et nous pouvons appliquer les mêmes concepts que nous avons appris précédemment à propos du déclenchement, de l'attachement, du détachement, du stoppage d'événement, etc...

Un élément important que vous devriez considérer est qu'il y a des événements qui seront déclenchés en ayant le même nom mais différents sujets, donc les vérifier dans l'objet événement est généralement requis dans chacune des fonctions qui sont attachées globalement pour éviter quelques bugs. Souvenez-vous qu'une extrême flexibilité implique une extrême complexité.

Examinez ce callback qui veut écouter tous les models beforeFind mais en réalité, il ne peut pas faire sa logique si le model est le caddie :

```
App::uses('CakeEventManager', 'Event');
CakeEventManager::instance()->attach('myCallback', 'Model.beforeFind');

public function myCallback($event) {
    if ($event->subject() instanceof Cart) {
```

```
        return;
    }
    return array('conditions' => ...);
}
```

Conclusion

Les événements sont une bonne façon de séparer les préoccupations dans votre application et rend les classes a la fois cohérentes et découplées des autres, néanmoins l'utilisation des événements n'est pas la solution à tous les problèmes. La plupart des applications n'auront pas réellement besoin de cette fonctionnalité, nous vous recommandons d'examiner d'autres options quand vous implémentez les callbacks comme l'utilisation des behaviors, des components et des helpers.

Gardez à l'esprit que beaucoup de pouvoir implique beaucoup de responsabilité, découpler vos classes par ce moyen signifie également que vous avez besoin d'effectuer une plus grande et meilleur intégration en testant votre code. Abuser de cet outil ne donnera pas à votre application une meilleure architecture, bien au contraire, cela sera beaucoup plus difficile à lire. En revanche si vous utilisez cet option à bon escient, seulement pour les choses dont vous avez besoin, Ce sera plus simple de travailler avec votre code, de le tester et de l'intégrer.

Lecture Supplémentaire

Collections Les Components, les Helpers, les Behaviors et les Tasks partagent tous une structure similaire et des comportements. Cake 2.0 fournit maintenant une API unifiée pour interagir avec les objets collections similaires. L'objet collection dans cakePHP vous donne un moyen uniforme d'interagir avec différentes sortes d'objets dans votre application.

Même si les exemples ci-dessous utiliseront des Components, le même comportement peut être envisagé pour les Helpers, Behaviors, et des Tasks en plus des components.

Charger et Décharger les objets Le chargement d'objets sur n'importe quelle collection peut être effectué en utilisant la méthode `load()` :

```
$this->Prg = $this->Components->load('Prg');
$this->Prg->process();
```

Au chargement du component, si il n'est pas chargé dans la collection, une nouvelle instance sera créée. Si le component est déjà chargé, une autre instance ne sera pas créée. Au chargement des components, vous pouvez aussi leurs fournir des configurations additionnelles :

```
$this->Cookie = $this->Components->load('Cookie', array('name' => 'sweet'));
```

Tout couple clé/valeur fourni sera passée au constructeur de Component. Une exception à cette règle est `className`. ClassName est une clé spéciale qui est utilisée pour créer des alias d'objets dans une collection. Ceci permet d'avoir des noms de component qui ne reflètent pas les noms de classes, ce qui peut être utile quand on étend les components du noyau :

```
$this->Auth = $this->Components->load('Auth', array('className' => 'MyCustomAuth'));
$this->Auth->user(); // Utilise réellement MyCustomAuth::user();
```

L'inverse du chargement d'un objet, est son déchargement. Les objets déchargés sont retirés de la mémoire, et n'auront pas de callbacks supplémentaires déclenchés sur eux :

```
$this->Components->unload('Cookie');
$this->Cookie->read(); // Fatal error.
```

Déclenchement de callbacks Les callbacks sont supportés par les collections d'objets. Quand une collection a un callback déclenché, cette méthode sera appelée sur tous les objets activés dans la collection. Vous pouvez passer des paramètres au boucle de callback comme ceci

```
$this->Behaviors->trigger('afterFind', array($this, $results, $primary));
```

Ci-dessus `$this` sera passé comme premier argument à toutes les méthodes afterFind des helpers. Il y a plusieurs options qui peuvent être utilisées pour contrôler comment les callbacks sont tués :

- `breakOn` Défini à la valeur ou aux valeurs pour lesquels vous voulez stopper la propagation. Peut être une valeur scalaire, ou un tableau de valeur à stopper. `False` par défaut.
- `break` Défini à true pour valider l'arrêt. Quand un déclancheur est cassé, la dernière valeur sera retournée. Si utilisé en combinaison avec `collectReturn` les résultats collectés seront retournés. ``False` par défaut.
- `collectReturn` Défini à true pour collecter le retour de chaque objet dans un tableau. Ce tableau de données retournées sera retourné depuis l'appel trigger(). `False` par défaut.
- `triggerDisabled` Déclenchera le callback sur tous les objets dans la collection même ceux qui sont non-activés. `False` par défaut.
- `modParams` Permet à chacun des objets auquel le callback à fait des demandes de modifier les paramètres de l'objet suivant. Paramétrer modParams en valeur entière vous permettra de modifier le paramètre avec cet index. N'importe quelle valeur non-nulle modifiera l'index de paramètre indiqué. `False` par défaut.

Effacer des boucles de callback En utilisant les options `break` et `breakOn` vous pouvez annuler une boucle de callback à mi-chemin semblable à interrompre la propagation événementielle en JavaScript

```
$this->Behaviors->trigger(
    'beforeFind',
    array($this, $query),
    array('break' => true, 'breakOn' => false)
);
```

Dans l'exemple ci-dessus, si n'importe quel behavior retourne `false` depuis sa méthode beforeFind, il n'y aura pas d'autres callback appelés. Le retour de `trigger()` sera false.

Activation et désactivation des objets Une fois qu'un objet est chargé dans une collection vous pourriez avoir besoin de le déactiver. Désactiver un objet dans une collection empêche aux futurs callbacks d'être tués sur l'objet à moins que l'option `triggerDisabled` soit utilisée :

```
// Désactive le Helper HTML
$this->Helpers->disable('Html');

// Ré-active le Helper plus tard
$this->Helpers->enable('Html');
```

Les objets désactivés peuvent toujours avoir leur méthodes et propriétés normales utilisées. La différence majeure entre un objet activé et désactivé se fait en regard des callbacks. Vous pouvez interroger une collection pour connaître les objets activés, ou vérifier si un objet spécifique est toujours activé en utilisant `enabled()` :

```
// Vérifie si oui ou on un Helper spécifique est activé.
$this->Helpers->enabled('Html');

// $enabled contiendra un tableau des helpers actuellement activés.
$enabled = $this->Helpers->enabled();
```

Behaviors (Comportements) Les behaviors (comportements) de Model sont une manière d'organiser certaines des fonctionnalités définies dans les models CakePHP. Ils nous permettent de séparer la logique qui ne doit pas être directement reliée à un model, mais qui nécessite d'être là. En offrant une simple, mais puissante, manière d'étendre les models, les behaviors nous permettent d'attacher des fonctionnalités aux models en définissant une simple variable de classe. C'est comme çà que les behaviors permettent de débarrasser les models de tout le "sur-poids" qui ne devrait pas faire partie du contrat métier qu'ils modèlent ou de ce qui est aussi nécessité par différents models et qui peut alors être extrapolé.

Par exemple, considérez un model qui nous donne accès à une table qui stocke des informations sur la structure d'un arbre hiérarchique. Supprimer, ajouter ou déplacer les nœuds dans l'arbre n'est pas aussi simple que d'effacer, d'insérer ou d'éditer les lignes d'une table. De nombreux enregistrements peuvent nécessiter une mise à jour suite au déplacement des éléments. Plutôt que de créer ces méthodes de manipulation d'arbre une fois par model de base (pour chaque model nécessitant cette fonctionnalité), nous pourrions simplement dire à notre model d'utiliser le behavior `TreeBehavior` ou, en des termes plus formels, nous dirions à notre model de se comporter comme un Arbre. On appelle cela attacher un behavior à un model. Avec une seule ligne de code, notre model CakePHP disposera d'un nouvel ensemble complet de méthodes lui permettant d'interagir avec la structure sous-jacente.

CakePHP contient déjà des behaviors pour les structures en arbre, les contenus traduits, les interactions par liste de contrôle d'accès, sans oublier les behaviors des contributeurs de la communauté déjà disponibles dans la Boulangerie (Bakery) CakePHP (http ://bakery.cakephp.org). Dans cette section nous couvrirons le schéma d'usage classique pour ajouter des behaviors aux models, l'utilisation des behaviors intégrés à CakePHP et la manière de créer nos propres behaviors.

Au final, les Behaviors sont Mixins (http ://en.wikipedia.org/wiki/Mixin) avec les callbacks.

Utiliser les Behaviors Les Behaviors sont attachés aux models grâce à la variable `$actsAs` des classes model :

```
class Category extends AppModel {
    public $actsAs = array('Tree');
}
```

Cette exemple montre comme un model Category pourrait être gérer dans une structure en arbre en utilisant le behavior Tree. Une fois qu'un behavior a été spécifié, utilisez les méthodes qu'il ajoute comme si elles avaient toujours existé et fait partie du model original :

```
// Définir ID
$this->Category->id = 42;

// Utiliser la méthode children() du behavior:
$kids = $this->Category->children();
```

Quelques behaviors peuvent nécessiter ou permettre des réglages quand ils sont attachés au model. Ici, nous indiquons à notre behavior Tree les noms des champs "left" et "right" de la table sous-jacente :

```
class Category extends AppModel {
    public $actsAs = array('Tree' => array(
        'left'  => 'left_node',
        'right' => 'right_node'
    ));
}
```

Nous pouvons aussi attacher plusieurs behaviors à un model. Il n'y aucune raison pour que, par exemple, notre model Category se comporte seulement comme un arbre, il pourrait aussi supporter l'internationalisation :

```
class Category extends AppModel {
    public $actsAs = array(
        'Tree' => array(
          'left'  => 'left_node',
          'right' => 'right_node'
        ),
        'Translate'
    );
}
```

Jusqu'à présent, nous avons ajouter les behaviors aux models en utilisant une variable de classe. Cela signifie que nos behaviors seront attachés à nos models de tout au long de leur durée vie. Pourtant, nous pourrions avoir besoin de "détacher" les behaviors des models à l'exécution. Considérons que dans notre précédent model Category, lequel agit comme un model Tree et Translate, nous ayons besoin pour quelque raison de le forcer à ne plus agir comme un model Translate :

```
// Détache un behavior de notre model :
$this->Category->Behaviors->unload('Translate');
```

Cela fera que notre model Categorie arrêtera dorénavant de se comporter comme un model Translate. Nous pourrions avoir besoin, sinon, de désactiver simplement le behavior Translate pour qu'il n'agisse pas sur les opérations normales de notre model : nos finds, nos saves, etc. En fait, nous cherchons à désactiver le behavior qui agit sur nos callbacks de model CakePHP. Au lieu de détacher le behavior, nous allons dire à notre model d'arrêter d'informer ses callbacks du behavior Translate :

```
// Empêcher le behavior de manipuler nos callbacks de model
$this->Category->Behaviors->disable('Translate');
```

Nous pourrions également avoir besoin de chercher si notre behavior manipule ces callbacks de model et si ce n'est pas le cas, alors de restaurer sa capacité à réagir avec eux :

```
// Si notre behavior ne manipule pas nos callbacks de model
if (!$this->Category->Behaviors->enabled('Translate')) {
    // Disons lui de le faire maintenant !
    $this->Category->Behaviors->enable('Translate');
}
```

De la même manière que nous pouvons détacher complètement un behavior d'un model à l'exécution, nous pouvons aussi attacher de nouveaux behaviors. Disons que notre model familier Category nécessite de se comporter comme un model Christmas, mais seulement le jour de Noël :

```
// Si nous sommes le 25 déc
if (date('m/d') === '12/25') {
    // Notre model nécessite de se comporter comme un model Christmas
    $this->Category->Behaviors->load('Christmas');
}
```

Nous pouvons aussi utiliser la méthode attach pour surcharger les réglages du behavior :

```
// Nous changerons un réglage de notre behavior déjà attaché
$this->Category->Behaviors->load('Tree', array('left' => 'new_left_node'));
```

Il y a aussi une méthode pour obtenir la liste des behaviors qui sont attachés à un model. Si nous passons le nom d'un behavior à une méthode, elle nous dira si ce behavior est attaché au model, sinon elle nous donnera la liste des behaviors attachés :

```
// Si le behavior Translate n'est pas attaché
if (!$this->Category->Behaviors->attached('Translate')) {
    // Obtenir la liste de tous les behaviors qui sont attachés au model
    $behaviors = $this->Category->Behaviors->attached();
}
```

Créer des Behaviors Les behaviors qui sont attachés aux Models voient leurs callbacks appelés automatiquement. Ces callbacks sont similaires à ceux qu'on trouve dans les Models : `beforeFind`, `afterFind`, `beforeSave`, `afterSave`, `beforeDelete`, `afterDelete` et `onError`. Voir *Méthodes Callback*.

Vos behaviors devront être placés dans `app/Model/Behavior`. Ils sont nommés en CamelCase et suffixés par `Behavior`, par ex. NomBehavior.php. Il est utile d'utiliser un behavior du coeur comme template quand on crée son propre behavior. Vous les trouverez dans `lib/Cake/Model/Behavior/`.

Chaque callback et behavior prend comme premier paramètre, une référence du model par lequel il est appelé.

En plus de l'implémentation des callbacks, vous pouvez ajouter des réglages par behavior et/ou par liaison d'un behavior au model. Des informations à propos des réglages spécifiques peuvent être trouvées dans les chapitres concernant les behaviors du cœur et leur configuration.

Voici un exemple rapide qui illustre comment les réglages peuvent êtres passés du model au behavior :

```
class Post extends AppModel {
    public $actsAs = array(
        'YourBehavior' => array(
            'option1_key' => 'option1_valeur'
        )
    );
}
```

Puisque les behaviors sont partagés à travers toutes les instances de model qui l'utilisent, une bonne pratique pour stocker les paramètres par nom d'alias/model qui utilise le behavior. La création des behaviors entraînera l'appel de leur méthode `setup()` :

```
public function setup(Model $Model, $settings = array()) {
    if (!isset($this->settings[$Model->alias])) {
        $this->settings[$Model->alias] = array(
            'option1_key' => 'option1_default_value',
            'option2_key' => 'option2_default_value',
            'option3_key' => 'option3_default_value',
        );
    }
    $this->settings[$Model->alias] = array_merge(
        $this->settings[$Model->alias], (array)$settings);
}
```

Créer les méthodes du behavior Les méthodes du Behavior sont automatiquement disponibles sur tout model qui 'act as' le behavior. Par exemple si vous avez :

```
class Duck extends AppModel {
    public $actsAs = array('Flying');
}
```

Vous seriez capable d'appeler les méthodes de `FlyingBehavior` comme si elles étaient des méthodes du model Duck. Quand on créer des méthodes d'un behavior, vous obtenez automatiquement une référence du model appelé en premier paramètre. Tous les autres paramètres fournis sont shifté one place to the right. Par exemple :

```
$this->Duck->fly('toronto', 'montreal');
```

Bien que cette méthode prenne deux paramètres, la méthode signature ressemblerait à cela :

```
public function fly(Model $Model, $from, $to) {
    // Faire quelque chose à la volée.
}
```

Gardez à l'esprit que les méthodes appelées dans un fashion `$this->doIt()` à partir de l'intérieur d'une méthode d'un behavior n'obtiendra pas le paramètre $model automatiquement annexé.

Méthodes mappées En plus de fournir des méthodes 'mixin', les behaviors peuvent aussi fournir des méthodes d'appariemment de formes (pattern matching). Les Behaviors peuvent aussi définir des méthodes mappées. Les méthodes mappées utilisent les pattern matching for method invocation. Cela vous permet de créer des méthodes du type `Model::findAllByXXX` sur vos behaviors. Les méthodes mappées ont besoin d'être déclarées dans votre tableau `$mapMethods` de behaviors. La signature de la méthode pour une méthode mappée est légèrement différente de celle d'une méthode mixin normal d'un behavior :

```
class MyBehavior extends ModelBehavior {
    public $mapMethods = array('/do(\w+)/' => 'doSomething');

    public function doSomething($model, $method, $arg1, $arg2) {
        debug(func_get_args());
        //faire quelque chose
    }
}
```

Ce qui est au-dessus mappera chaque méthode `doXXX()` appélé vers le behavior. Comme vous pouvez le voir, le model est toujours le premier paramètre, mais le nom de la méthode appelée sera le deuxième paramètre. Cela vous permet de munge le nom de la méthode pour des informations supplémentaires, un peu comme `Model::findAllByXX`. Si le behavior du dessus est attaché à un model, ce qui suit arrivera :

```
$model->doReleaseTheHounds('homer', 'lenny');

// sortira
'ReleaseTheHounds', 'homer', 'lenny'
```

Callbacks du Behavior Les Behaviors d'un Model peuvent définir un nombre de callbacks qui sont déclenchés before les callbacks du model du même nom. Les callbacks du Behavior vous permettent de capturer des évènements dans les models attachés et d'augmenter les paramètres ou accoler dans un behavior supplémentaire.

Les callbacks disponibles sont :
- `beforeValidate`
- `beforeFind`
- `afterFind`
- `beforeSave`
- `afterSave`
- `beforeDelete`
- `afterDelete`

Créer un callback du behavior

class `ModelBehavior`

Les callbacks d'un behavior d'un model sont définis comme de simples méthodes dans votre classe de behavior. Un peu comme les méthodes classiques du behavior, ils reçoivent un paramètre `$Model` en premier argument. Ce paramètre est le model pour lequel la méthode du behavior a été invoquée.

`ModelBehavior::`**`setup`**(*Model $Model*, *array $settings = array()*)

Appelé quand un behavior est attaché à un model. Les paramètres viennent de la propriété `$actsAs` du model attaché.

`ModelBehavior::cleanup`(*Model $Model*)
Appelé quand un behavior est détaché d'un model. La méthode de base retire les paramètres du model basées sur `$model->alias`. Vous pouvez écraser cette méthode et fournir une fonctionnalité personnalisée nettoyée.

`ModelBehavior::beforeFind`(*Model $Model*, *array $query*)
Si le beforeFind du behavior retourne false, cela annulera le find(). Retourner un tableau augmentera les paramètres de requête utilisés pour l'opération find.

`ModelBehavior::afterFind`(*Model $Model*, *mixed $results*, *boolean $primary*)
Vous pouvez utiliser le afterFind pour augmenter les résultats d'un find. La valeur retournée sera passée en résultats soit au behavior suivant dans la chaîne, soit au afterFind du model.

`ModelBehavior::beforeDelete`(*Model $Model*, *boolean $cascade = true*)
Vous pouvez retourner false d'un beforeDelete d'un behavior pour annuler la suppression. Retourne true pour autoriser la suite.

`ModelBehavior::afterDelete`(*Model $Model*)
Vous pouvez utiliser afterDelete pour effectuer des opérations de nettoyage liées à votre behavior.

`ModelBehavior::beforeSave`(*Model $Model*)
Vous pouvez retourner false d'un beforeSave d'un behavior pour annuler la sauvegarde. Retourner true pour permettre de continuer.

`ModelBehavior::afterSave`(*Model $Model*, *boolean $created*)
Vous pouvez utiliser afterSave pour effectuer des opérations de nettoyage liées au behavior. $created sera à true quand un enregistrement sera crée, et à false quand un enregistrement sera mis à jour.

`ModelBehavior::beforeValidate`(*Model $Model*)
Vous pouvez utiliser beforeValidate pour modifier un tableau de validation de model ou gérer tout autrre logique de pré-validation. Retourner false d'un callback beforeValidate annulera la validation et entraînera son echec.

Components (Composants) Les Components (Composants) sont des regroupements de logique applicative qui sont partagés entre les controllers. Si vous vous surprenez à vouloir copier et coller des choses entre vos controllers, alors vous devriez envisager de regrouper plusieurs fonctionnalités dans un Component.

CakePHP est également livré avec un fantastique ensemble de components, que vous pouvez utiliser pour vous aider :
- Sécurité
- Sessions
- Listes de contrôle d'accès (ACL)
- Emails
- Cookies
- Authentification
- Traitement de requêtes
- Pagination

Chacun de ces components d'origine est détaillé dans des chapitres spécifiques. Pour l'heure, nous allons vous montrer comment créer vos propres components. La création de components vous permet de garder le code de vos controllers propres et vous permet de réutiliser du code entre vos projets.

Configuration des Components De nombreux components du cœur nécessitent une configuration. Quelques exemples : *Authentification*, *Cookie* et *EmailComponent*. Toute configuration pour ces components, et pour les components en général, se fait dans le tableau des `$components` de la méthode `beforeFilter()` de vos controllers :

```
class PostsController extends AppController {
    public $components = array(
        'Auth' => array(
            'authorize' => array('controller'),
            'loginAction' => array('controller' => 'users', 'action' => 'login')
        ),
        'Cookie' => array('name' => 'CookieMonster')
    );
```

Serait un exemple de configuration d'un component avec le tableau `$components`. Tous les components du coeur permettent aux paramètres d'être configurés dans la méthode de votre controller `beforeFilter()`. C'est utile quand vous avez besoin d'assigner les résultats d'une fonction à la propriété d'un component. Ceci peut aussi être exprimé comme ceci :

```
public function beforeFilter() {
    $this->Auth->authorize = array('controller');
    $this->Auth->loginAction = array('controller' => 'users', 'action' => 'login');

    $this->Cookie->name = 'CookieMonster';
}
```

C'est possible, cependant, que le component nécessite certaines options de configuration avant que le controller `beforeFilter()` soit lancé. Pour cela, certains components permettent aux options de configuration d'être définies dans le tableau `$components` :

```
public $components = array(
    'DebugKit.Toolbar' => array('panels' => array('history', 'session'))
);
```

Consultez la documentation appropriée pour connaître les options de configuration que chaque component fournit.

Un paramètre commun à utiliser est l'option `className`, qui vous autorise les alias des components. Cette fonctionnalité est utile quand vous voulez remplacer `$this->Auth` ou une autre référence de Component commun avec une implémentation sur mesure :

```
// app/Controller/PostsController.php
class PostsController extends AppController {
    public $components = array(
        'Auth' => array(
            'className' => 'MyAuth'
        )
    );
}

// app/Controller/Component/MyAuthComponent.php
App::uses('AuthComponent', 'Controller/Component');
class MyAuthComponent extends AuthComponent {
```

```
    // Ajouter votre code pour surcharger le AuthComponent du coeur
}
```

Ce qu'il y a au-dessous donnerait un *alias* `MyAuthComponent` à `$this->Auth` dans vos controllers.

Note : Faire un alias à un component remplace cette instance n'importe où où le component est utilisé, en incluant l'intérieur des autres Components.

Utiliser les Components Une fois que vous avez inclu quelques components dans votre controller, les utiliser est très simple. Chaque component que vous utilisez est enregistré comme propriété dans votre controller. Si vous avez chargé la `SessionComponent` et le `CookieComponent` dans votre controller, vous pouvez y accéder comme ceci :

```
class PostsController extends AppController {
    public $components = array('Session', 'Cookie');

    public function delete() {
        if ($this->Post->delete($this->request->data('Post.id')) {
            $this->Session->setFlash('Post deleted.');
            $this->redirect(array('action' => 'index'));
        }
    }
```

Note : Puisque les Models et les Components sont tous deux ajoutés aux controllers en tant que propriété, ils partagent le même 'espace de noms'. Assurez vous de ne pas donner le même nom à un component et à un model.

Charger les components à la volée Vous n'avez parfois pas besoin de rendre le component accessible sur chaque action. Dans ce cas là, vous pouvez charger à la volée en utilisant la *Component Collection*. A partir de l'intérieur d'un controller, vous pouvez faire comme ce qui suit :

```
$this->OneTimer = $this->Components->load('OneTimer');
$this->OneTimer->getTime();
```

Note : Gardez à l'esprit que le chargement d'un component à la volée ne va pas appeler la méthode initialize. Si le component que vous appelez a cette méthode, vous devrez l'appeler manuellement après le chargement.

Callbacks des Components Les components vous offrent aussi quelques callbacks durant leur cycle de vie qui vous permettent d'augmenter le cycle de la requête. Allez voir l'api *API de Component* pour plus d'informations sur les callbacks possibles des components.

Créer un Component Supposons que notre application en ligne ait besoin de réaliser une opération mathématique complexe dans plusieurs sections différentes de l'application. Nous pourrions créer un component pour héberger cette logique partagée afin de l'utiliser dans plusieurs controllers différents.

La première étape consiste à créer un nouveau fichier et une classe pour le component. Créez le fichier dans `/app/Controller/Component/MathComponent.php`. La structure de base pour le component ressemblerait à quelque chose comme ça

```
class MathComponent extends Component {
    public function faireDesOperationsComplexes($montant1, $montant2) {
        return $montant1 + $montant2;
    }
}
```

Note : Tous les components comme Math doivent étendre `Component`. Ne pas le faire vous enverra une exception.

Inclure votre component dans vos controllers Une fois notre component terminé, nous pouvons l'utiliser au sein des controllers de l'application en plaçant son nom (sans la partie "Component") dans le tableau `$components` du controller. Le controller sera automatiquement pourvu d'un nouvel attribut nommé d'après le component, à travers lequel nous pouvons accéder à une instance de celui-ci :

```
/* Rend le nouveau component disponible par $this->Math
ainsi que le component standard $this->Session */
public $components = array('Math', 'Session');
```

Les Components déclarés dans `AppController` seront fusionnés avec ceux déclarés dans vos autres controllers. Donc il n'y a pas besoin de re-déclarer le même component deux fois.

Quand vous incluez des Components dans un Controller, vous pouvez aussi déclarer un ensemble de paramètres qui seront passés à la méthode initialize() du Component. Ces paramètres peuvent alors être pris en charge par le Component :

```
public $components = array(
    'Math' => array(
        'precision' => 2,
        'generateurAleatoire' => 'srand'
    ),
    'Session', 'Auth'
);
```

L'exemple ci-dessus passerait le tableau contenant "precision" et "generateurAleatoire" comme second paramètre au `MathComponent::__construct()`. Par convention, tout paramètre passé qui est aussi une propriété publique sur votre component aura la valeur basée sur ces paramètres.

Utiliser d'autres Components dans votre Component Parfois un de vos components a besoin d'utiliser un autre component. Dans ce cas, vous pouvez inclure d'autres components dans votre component exactement de la même manière que dans vos controllers - en utilisant la variable `$components` :

```
// app/Controller/Component/CustomComponent.php
class CustomComponent extends Component {
    // l'autre component que votre component utilise
    public $components = array('Existing');

    public function initialize($controller) {
        $this->Existing->foo();
    }

    public function bar() {
        // ...
   }
}

// app/Controller/Component/ExistingComponent.php
class ExistingComponent extends Component {

    public function initialize($controller) {
        $this->Parent->bar();
    }

    public function foo() {
        // ...
    }
}
```

Notez qu'au contraire d'un component inclu dans un controller, aucun callback ne sera attrapé pour un component inclu dans un component.

API de Component

class `Component`

La classe de base de Component vous offre quelques méthodes pour le chargement facile des autres Components à travers `ComponentCollection` comme nous l'avons traité avec la gestion habituelle des paramètres. Elle fournit aussi des prototypes pour tous les callbacks des components.

`Component::__construct`(*ComponentCollection $collection*, *$settings = array()*)

Les Constructeurs pour la classe de base du component. Tous les `$parametres` qui sont aussi des propriétés publiques, vont avoir leurs valeurs changées pour matcher avec les valeurs de `$settings`.

Les Callbacks

`Component::initialize`(*Controller $controller*)

La méthode initialize est appelée avant la méthode du controller beforeFilter.

`Component::startup`(*Controller $controller*)

La méthode startup est appelée après la méthode du controller beforeFilter mais avant que le controller n'exécute l'action prévue.

`Component::beforeRender`(*Controller $controller*)

La méthode beforeRender est appelée après que le controller exécute la logique de l'action requêtée, mais avant le rendu de la vue et le layout du controller.

`Component::shutdown`(*Controller $controller*)
La méthode shutdown est appelée avant que la sortie soit envoyée au navigateur.

`Component::beforeRedirect`(*Controller $controller*, *$url*, *$status=null*, *$exit=true*)
La méthode beforeRedirect est invoquée quand la méthode de redirection du controller est appelée, mais avant toute action qui suit. Si cette méthode retourne false, le controller ne continuera pas de rediriger la requête. Les variables $url, $status et $exit ont la même signification que pour la méthode du controller. Vous pouvez aussi retourner une chaîne de caractère qui sera interpretée comme une url pour rediriger ou retourner un array associatif avec la clé 'url' et éventuellement 'status' et 'exit'.

Helpers (Assistants) Les Helpers (Assistants) sont des classes comme les components, pour la couche de présentation de votre application. Ils contiennent la logique de présentation qui est partagée entre plusieurs vues, éléments ou layouts. Ce chapitre vous montrera comment créer vos propres helpers et soulignera les tâches basiques que les helpers du cœur de CakePHP peuvent vous aider à accomplir.

CakePHP dispose d'un nombre de helpers qui aident à la création des vues. Ils aident à la création de balises bien-formées (y compris les formulaires), aident à la mise en forme du texte, les durées et les numéros, et peuvent même accélérer la fonctionnalité Ajax. Pour plus d'informations sur les helpers inclus dans CakePHP, allez voir *Helpers (Assistants)*.

Utiliser et configurer les Helpers Vous activez les helpers (assistants) dans CakePHP, en faisant "prendre conscience" à un controller qu'ils existent. Chaque controller a une propriété `$helpers`, qui liste les helpers disponibles dans la vue. Pour activer un helper dans votre vue, ajoutez son nom au tableau `$helpers` du controller :

```
class BakeriesController extends AppController {
    public $helpers = array('Form', 'Html', 'Js', 'Time');
}
```

L'ajout des helpers depuis les plugins utilise la *syntaxe de plugin* utilisée partout ailleurs dans CakePHP :

```
class BakeriesController extends AppController {
    public $helpers = array('Blog.Comment');
}
```

Vous pouvez aussi ajouter les helpers depuis une action, dans ce cas, ils seront uniquement accessibles pour cette action et aucune autre dans le controller. Ceci économise de la puissance de calcul pour les autres actions qui n'utilisent pas le helper, tout en permettant de conserver le controller mieux organisé :

```
class BakeriesController extends AppController {
    public function bake {
        $this->helpers[] = 'Time';
    }
    public function mix {
        // Le Helper Time n'est pas chargé ici et n'est par conséquent
        pas disponible
    }
}
```

Si vous avez besoin d'activer un helper pour tous les controllers, ajoutez son nom dans le tableau `$helpers` du fichier `/app/Controller/AppController.php` (à créer si pas présent). N'oubliez pas d'inclure les helpers par défaut Html et Form :

```
class AppController extends Controller {
    public $helpers = array('Form', 'Html', 'Js', 'Time');
}
```

Vous pouvez passer des options dans les helpers. Ces options peuvent être utilisées pour définir les valeurs d'attributs ou modifier le comportement du helper :

```
class AwesomeHelper extends AppHelper {
    public function __construct(View $view, $settings = array()) {
        parent::__construct($view, $settings);
        debug($options);
    }
}

class AwesomeController extends AppController {
    public $helpers = array('Awesome' => array('option1' => 'valeur1'));
}
```

Depuis 2.3 les options sont fusionnées avec la propriété `Helper::$settings` du helper.

Une configuration courante est d'utiliser l'option `className`, qui vous permet de créer des helpers alias dans vos vues. Cette fonctionnalité est utile quand vous voulez remplacer `$this->Html` ou tout autre Helper de référence avec une mise en oeuvre personnalisée :

```
// app/Controller/PostsController.php
class PostsController extends AppController {
    public $helpers = array(
        'Html' => array(
            'className' => 'MyHtml'
        )
    );
}

// app/View/Helper/MyHtmlHelper.php
App::uses('HtmlHelper', 'View/Helper');
class MyHtmlHelper extends HtmlHelper {
    // Ajouter votre code pour écraser le HtmlHelper du coeur
}
```

Ce qui est au-dessus ferait un *alias* de `MyHtmlHelper` vers `$this->Html` dans vos vues.

Note : Faire un alias d'un helper remplace cette instance partout où le helper est utilisé, y compris dans les autres Helpers.

Astuce : Faire un alias des Helpers Html ou Session pendant que vous utilisez le coeur de PagesController ne fonctionnera pas. Il est préférable de copier `lib/Cake/Controller/PagesController.php` dans le dossier `app/Controller/`.

L'utilisation des configurations du helper vous permet de configurer de manière déclarative vos helpers et de garder la logique de configuration de vos actions des controllers. Si vous avez des options de configuration qui ne peuvent pas être inclues comme des parties de déclaration de classe, vous pouvez les définir dans le callback beforeRender de votre controller :

```
class PostsController extends AppController {
    public function beforeRender() {
        parent::beforeRender();
        $this->helpers['CustomStuff'] = $this->_getCustomStuffSettings();
    }
}
```

Utiliser les Helpers Une fois que vous avez configuré les helpers que vous souhaitiez utiliser, dans votre controller, chaque helper est exposé en propriété publique dans la vue. Par exemple, si vous utilisiez `HtmlHelper`, vous seriez capable d'y accéder en faisant ce qui suit :

```
echo $this->Html->css('styles');
```

Ce qui est au-dessus appelerait la méthode `css` du HtmlHelper. Vous pouvez accéder à n'importe quel helper chargé en utilisant `$this->{$helperName}`. Il peut venir un temps où vous aurez besoin de charger dynamiquement un helper à partir d'une vue. Vous pouvez utiliser la vue du `HelperCollection` pour le faire :

```
$mediaHelper = $this->Helpers->load('Media', $mediaSettings);
```

Le HelperCollection est une *collection* et supporte l'API collection utilisée partout ailleurs dans CakePHP.

Méthodes de Callback Les Helpers disposent de plusieurs callbacks qui vous permettent d'augmenter le processus de rendu de vue. Allez voir la documentation de *API de Helper* et *Collections* pour plus d'informations.

Créer des Helpers Si un helper du coeur (ou l'un présenté sur github ou dans la Boulangerie) ne correspond pas à vos besoins, les helpers sont faciles à créer.

Mettons que nous voulions créer un helper, qui pourrait être utilisé pour produire un lien CSS, façonné spécialement selon vos besoins, à différents endroits de votre application. Afin de trouver une place à votre logique dans la structure de helper existante dans CakePHP, vous devrez créer une nouvelle classe dans `/app/View/Helper`. Appelons notre assistant LienHelper. Le fichier de la classe PHP devrait ressembler à quelque chose comme ceci :

```
/* /app/View/Helper/LienHelper.php */
App::uses('AppHelper', 'View/Helper');

class LienHelper extends AppHelper {
    public function lancerEdition($titre, $url) {
        // La logique pour créer le lien spécialement formaté se place
        ici...
```

```
    }
}
```

Note : Les Helpers doivent étendre soit `AppHelper` soit `Helper` ou implémenter tous les callbacks dans *API de Helper*.

Inclure d'autres Helpers Vous souhaitez peut-être utiliser quelques fonctionnalités déjà existantes dans un autre helper. Pour faire cela, vous pouvez spécifier les helpers que vous souhaitez utiliser avec un tableau `$helpers`, formaté comme vous le feriez dans un controller :

```
/* /app/View/Helper/LienHelper.php (Utilisant d'autres helpers) */
App::uses('AppHelper', 'View/Helper');

class LienHelper extends AppHelper {
    public $helpers = array('Html');

    public function lancerEdition($titre, $url) {
        // Utilisation du helper HTML pour sortir une donnée formatée

        $link = $this->Html->link($titre, $url, array('class' => 'edit'));

        return '<div class="editOuter">' . $link . '</div>';
    }
}
```

Utiliser votre Helper Une fois que vous avez créez votre helper et l'avez placé dans `/app/View/Helper/`, vous serez capable de l'inclure dans vos controllers en utilisant la variable spéciale `$helpers` :

```
class PostsController extends AppController {
    public $helpers = array('Lien');
}
```

Une fois que votre controller est au courant de cette nouvelle classe, vous pouvez l'utiliser dans vos vues en accédant à un objet nommé d'après le helper :

```
<!-- fait un lien en utilisant le nouveau helper -->
<?php echo $this->Lien->lancerEdition('Changer cette recette', '/recipes/edit/5'); ?>
```

Créer des fonctionnalités à vos Helpers Tous les helpers étendent une classe spéciale, AppHelper (comme les models étendent AppModel et les controllers étendent AppController). Pour créer une fonctionnalité disponible pour tous les helpers, créez `/app/View/Helper/AppHelper.php` :

```
App::uses('Helper', 'View');

class AppHelper extends Helper {
    public function customMethod () {
```

```
    }
}
```

API de Helper

class `Helper`

La classe de base pour les Helpers. Elle fournit un nombre de méthodes utiles et des fonctionnalités pour le chargement d'autres helpers.

`Helper::webroot`(*$file*)

Décide du nom de fichier du webroot de l'application. Si un thème est actif et que le fichier existe dans le webroot du thème courant, le chemin du fichier du thème sera retourné.

`Helper::url`(*$url*, *$full = false*)

Génère une HTML escaped URL, qui délégue à `Router::url()`.

`Helper::value`(*$options = array()*, *$field = null*, *$key = 'value'*)

Récupère la valeur pour un nom d'input donné.

`Helper::domId`(*$options = null*, *$id = 'id'*)

Génère une valeur id en CamelCase pour le champ sélectionné courant. Ecraser cette méthode dans votre AppHelper vous permettra de changer la façon dont CakePHP génére les attributs ID.

Callbacks

`Helper::beforeRenderFile`(*$viewFile*)

Est appelé avant que tout fichier de vue soit rendu. Cela inclut les eléments, le vues, les vues parentes et les layouts.

`Helper::afterRenderFile`(*$viewFile*, *$content*)

Est appelé après que tout fichier de vue est rendu. Cela inclut les eléments, le vues, les vues parentes et les layouts. Un callback peut modifier et retourner `$content` pour changer la manière dont le contenu rendu est affiché dans le navigateur.

`Helper::beforeRender`(*$viewFile*)

La méthode beforeRender est appelé après la méthode beforeRender du controller, mais avant les rendus du controller de la vue et du layout Reçoit le fichier à rendre en argument.

`Helper::afterRender`(*$viewFile*)

Est appelé après que la vue est rendu, mais avant que le rendu du layout ait commencé.

`Helper::beforeLayout`(*$layoutFile*)

Est appelé avant que le rendu du layout commence. Reçoit le nom du fichier layout en argument.

`Helper::afterLayout`(*$layoutFile*)

Est appelé après que le rendu du layout est fini. Reçoit le nom du fichier layout en argument.

Helpers du coeur

CacheHelper Utilisé par le coeur pour mettre en cache le contenu de la vue.

FormHelper Créé les formulaires HTML et les éléments du formulaire qui gèrent eux-mêmes les problèmes de validation.

HTMLHelper Méthodes bien pratiques pour des balises bien formatées. Les images, les liens, les tables, les balises d'en-tête etc ....

JSHelper Utilisé pour créer du Javascript compatible avec de nombreuses librairies Javascript.

NumberHelper Formate les nombres et les monnaies.

Paginator Pagination à partir des données de Modèles et tri.

RSS Méthodes bien pratiques pour la sortie de contenu RSS et de données XML.

SessionHelper Accès pour la lecture des valeurs de session dans les vues.

TextHelper Mise en lien intelligente, mise en évidence, truchement intelligent des mots.

TimeHelper Détection de proximité (Est-ce l'année prochaine ?), formatage sympa des chaînes de caractère (Today, 10 :30 am) et conversion entre les zones de temps.

Collections

Les Components, les Helpers, les Behaviors et les Tasks partagent tous une structure similaire et des comportements. Cake 2.0 fournit maintenant une API unifiée pour interagir avec les objets collections similaires. L'objet collection dans cakePHP vous donne un moyen uniforme d'interagir avec différentes sortes d'objets dans votre application.

Même si les exemples ci-dessous utiliseront des Components, le même comportement peut être envisagé pour les Helpers, Behaviors, et des Tasks en plus des components.

Charger et Décharger les objets

Le chargement d'objets sur n'importe quelle collection peut être effectué en utilisant la méthode `load()` :

```
$this->Prg = $this->Components->load('Prg');
$this->Prg->process();
```

Au chargement du component, si il n'est pas chargé dans la collection, une nouvelle instance sera créée. Si le component est déjà chargé, une autre instance ne sera pas créée. Au chargement des components, vous pouvez aussi leurs fournir des configurations additionnelles :

```
$this->Cookie = $this->Components->load('Cookie', array('name' => 'sweet'));
```

Tout couple clé/valeur fourni sera passée au constructeur de Component. Une exception à cette règle est `className`. ClassName est une clé spéciale qui est utilisée pour créer des alias d'objets dans une collection. Ceci permet d'avoir des noms de component qui ne reflètent pas les noms de classes, ce qui peut être utile quand on étend les components du noyau :

```
$this->Auth = $this->Components->load('Auth', array('className' => 'MyCustomAuth'));
$this->Auth->user(); // Utilise réellement MyCustomAuth::user();
```

L'inverse du chargement d'un objet, est son déchargement. Les objets déchargés sont retirés de la mémoire, et n'auront pas de callbacks supplémentaires déclenchés sur eux :

```
$this->Components->unload('Cookie');
$this->Cookie->read(); // Fatal error.
```

Déclenchement de callbacks

Les callbacks sont supportés par les collections d'objets. Quand une collection a un callback déclenché, cette méthode sera appelée sur tous les objets activés dans la collection. Vous pouvez passer des paramètres au boucle de callback comme ceci

```
$this->Behaviors->trigger('afterFind', array($this, $results, $primary));
```

Ci-dessus `$this` sera passé comme premier argument à toutes les méthodes afterFind des helpers. Il y a plusieurs options qui peuvent être utilisées pour contrôler comment les callbacks sont tués :

- `breakOn` Défini à la valeur ou aux valeurs pour lesquels vous voulez stopper la propagation. Peut être une valeur scalaire, ou un tableau de valeur à stopper. `False` par défaut.
- `break` Défini à true pour valider l'arrêt. Quand un déclancheur est cassé, la dernière valeur sera retournée. Si utilisé en combinaison avec `collectReturn` les résultats collectés seront retournés. ``False` par défaut.
- `collectReturn` Défini à true pour collecter le retour de chaque objet dans un tableau. Ce tableau de données retournées sera retourné depuis l'appel trigger(). `False` par défaut.
- `triggerDisabled` Déclenchera le callback sur tous les objets dans la collection même ceux qui sont non-activés. `False` par défaut.
- `modParams` Permet à chacun des objets auquel le callback à fait des demandes de modifier les paramètres de l'objet suivant. Paramétrer modParams en valeur entière vous permettra de modifier le paramètre avec cet index. N'importe quelle valeur non-nulle modifiera l'index de paramètre indiqué. `False` par défaut.

Effacer des boucles de callback En utilisant les options `break` et `breakOn` vous pouvez annuler une boucle de callback à mi-chemin semblable à interrompre la propagation événementielle en JavaScript

```
$this->Behaviors->trigger(
    'beforeFind',
    array($this, $query),
    array('break' => true, 'breakOn' => false)
);
```

Dans l'exemple ci-dessus, si n'importe quel behavior retourne `false` depuis sa méthode beforeFind, il n'y aura pas d'autres callback appelés. Le retour de `trigger()` sera false.

Activation et désactivation des objets

Une fois qu'un objet est chargé dans une collection vous pourriez avoir besoin de le déactiver. Désactiver un objet dans une collection empêche aux futurs callbacks d'être tués sur l'objet à moins que l'option `triggerDisabled` soit utilisée :

```
// Désactive le Helper HTML
$this->Helpers->disable('Html');

// Ré-active le Helper plus tard
$this->Helpers->enable('Html');
```

Les objets désactivés peuvent toujours avoir leur méthodes et propriétés normales utilisées. La différence majeure entre un objet activé et désactivé se fait en regard des callbacks. Vous pouvez interroger une collection pour connaître les objets activés, ou vérifier si un objet spécifique est toujours activé en utilisant `enabled()` :

```
// Vérifie si oui ou on un Helper spécifique est activé.
$this->Helpers->enabled('Html');

// $enabled contiendra un tableau des helpers actuellement activés.
$enabled = $this->Helpers->enabled();
```

Behaviors (Comportements)

Les behaviors ajoutent des fonctionnalités supplémentaires à vos models. CakePHP offre un nombre de behaviors integrés tels que `TreeBehavior` et `ContainableBehavior`.

Pour en apprendre plus sur la création et l'utilisation des behaviors, lire la section sur *Behaviors (Comportements)*.

Behaviors (Comportements)

ACL

class `AclBehavior`

Le behavior Acl fournit une solution pour intégrer sans souci un model dans votre système ACL. Il peut créer à la fois les AROs et les ACOs de manière transparente.

Pour utiliser le nouveau behavior, vous pouvez l'ajouter à la propriété $actsAs de votre model. Quand vous l'ajoutez au tableau actsAs, vous choisissez de créer l'entrée Acl correspondante comme un ARO ou un ACO. Par défaut, cela crée des AROs :

```
class User extends AppModel {
    public $actsAs = array('Acl' => array('type' => 'requester'));
}
```

Ceci attacherait le behavior Acl en mode ARO. Pour attacher le behavior ACL en mode ACO, utilisez

```
class Post extends AppModel {
    public $actsAs = array('Acl' => array('type' => 'controlled'));
}
```

Pour les models d'user et de group il est fréquent d'avoir à la fois les noeuds ACO et ARO, pour permettre cela utilisez :

```
class User extends AppModel {
    public $actsAs = array('Acl' => array('type' => 'both'));
}
```

Vous pouvez aussi attacher le behavior à la volée, comme ceci

```
$this->Post->Behaviors->attach('Acl', array('type' => 'controlled'));
```

Modifié dans la version 2.1 : Vous pouvez maintenant attacher en toute sécurité le behavior Acl (AclBehavior) à votre Appmodel. Aco, Aro et Noeud Acl (AclNode) sont dorénavant des extensions du Model et non plus de l'AppModel, ceci pouvait causer une boucle infinie. Si pour certaines raisons, votre application est dépendante de l'utilisation des models comme extension de l'AppModel, alors copiez Le Noeud Acl (AclNode) dans votre application et étendez à nouveau AppModel.

Utiliser le behavior Acl

La plupart des tâches du behavior Acl sont réalisées de façon transparente, dans le callback afterSave() de votre model. Cependant, son utilisation nécessite que votre Model ait une méthode parentNode() définie. Ceci est utilisé par le behavior Acl, pour déterminer les relations parent->enfant. Une méthode parentNode() de model doit retourner null ou une référence au Model parent :

```
public function parentNode() {
    return null;
}
```

Si vous voulez définir un nœud ACO ou ARO comme parent pour votre Model, parentNode() doit retourner l'alias du nœud ACO ou ARO :

```
public function parentNode() {
    return 'root_node';
}
```

Voici un exemple plus complet. Utilisons un model d'exemple User, avec User belongsTo Group :

```
public function parentNode() {
    if (!$this->id && empty($this->data)) {
        return null;
    }
    $data = $this->data;
    if (empty($this->data)) {
        $data = $this->read();
    }
    if (!$data['User']['group_id']) {
        return null;
    } else {
        return array('Group' => array('id' => $data['User']['group_id']));
    }
}
```

Dans l'exemple ci-dessus, le retour est un tableau qui ressemble aux résultats d'un find de model. Il est important d'avoir une valeur d'id définie ou bien la relation parentNode échouera. Le behavior Acl utilise ces données pour construire son arborescense.

node()

Le Behavior Acl vous permet aussi de récupérer le nœud Acl associé à un enregistrement de model. Après avoir défini $model->id. Vous pouvez utiliser $model->node() pour récupérer le nœud Acl associé.

Vous pouvez aussi récupérer le nœud Acl de n'importe quelle ligne, en passant un tableau de données en paramètre :

```
$this->User->id = 1;
$noeud = $this->User->node();

$user = array('User' => array(
    'id' => 1
));
$noeud = $this->User->node($user);
```

Ces deux exemples retourneront la même information de nœud Acl.

Si vous avez paramétré le behavior Acl (AclBehavior) pour créer à la fois les noeuds ARO et ACO, vous devez spécifier quel type de noeud vous desirez :

```
$this->User->id = 1;
$noeud = $this->User->node(null, 'Aro');

$user = array('User' => array(
    'id' => 1
));
$noeud = $this->User->node($user, 'Aro');
```

Containable

class `ContainableBehavior`

Une nouvelle intégration au coeur de CakePHP 1.2 est le Behavior "Containable" `ContainableBehavior`. Ce behavior vous permet de filtrer et limiter les opérations de récupération de données "find". Utiliser Containable vous aidera a réduire l'utilisation inutile de votre base de données et augmentera la vitesse et la plupart des performances de votre application. La classe vous aidera aussi a chercher et filtrer vos données pour vos utilisateurs d'une façon propre et cohérente.

Le behavior "Containable" vous permet de rationaliser et simplifier les opérations de construction du model. Il agit en modifiant temporairement ou définitivement les associations de vos models. Il fait cela en utilisant des "containements" pour génerer une série d'appels `bindModel` et `unbindModel`. Étant donné que Containable modifie seulement les relations déjà existantes, il ne vous permettra pas de restreindre les résultats pour des associations distantes. Pour cela, vous devriez voir les *Tables jointes*.

Pour utiliser le nouveau behavior, vous pouvez l'ajouter à la propriété $actAs de votre model :

```
class Post extends AppModel {
    public $actsAs = array('Containable');
}
```

Vous pouvez aussi attacher le behavior à la volée :

```
$this->Post->Behaviors->attach('Containable');
```

Utilisation de Containable

Pour voir comment Containable fonctionne, regardons quelques exemples. Premièrement, nous commencerons avec un appel find() sur un model nommé Post. Disons que ce Post a plusieurs (hasMany) Commentaire, et Post a et appartient à plusieurs (hasAndBelongsToMany) Tag. La quantité de données récupérées par un appel find() normal est assez étendue

```
debug($this->Post->find('all'));

[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [titre] => Premier article
                    [contenu] => aaa
                    [created] => 2008-05-18 00:00:00
                )
            [Commentaire] => Array
                (
                    [0] => Array
                        (
                            [id] => 1
                            [post_id] => 1
                            [auteur] => Daniel
                            [email] => dan@example.com
                            [siteweb] => http://example.com
                            [commentaire] => Premier commentaire
                            [created] => 2008-05-18 00:00:00
                        )
                    [1] => Array
                        (
                            [id] => 2
                            [post_id] => 1
                            [auteur] => Sam
                            [email] => sam@example.net
                            [siteweb] => http://example.net
                            [commentaire] => Second commentaire
                            [created] => 2008-05-18 00:00:00
                        )
                )
            [Tag] => Array
                (
```

```
                    [0] => Array
                        (
                            [id] => 1
                            [name] => A
                        )
                    [1] => Array
                        (
                            [id] => 2
                            [name] => B
                        )
                )
        )
[1] => Array
        (
            [Post] => Array
                (...
```

Pour certaines interfaces de votre application, vous pouvez ne pas avoir besoin d'autant d'information depuis le model Post. Le `Behavior containable` permet de reduire ce que le find() retourne.

Par exemple, pour ne récupérer que les informations relative au post vous pouvez faire cela :

```
$this->Post->contain();
$this->Post->find('all');
```

Vous pouvez utiliser la magie de "Containable" à l'interieur d'un appel find() :

```
$this->Post->find('all', array('contain' => false));
```

Après avoir fait cela, vous vous retrouvez avec quelque chose de plus concis :

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [titre] => Premier article
                    [contenu] => aaa
                    [created] => 2008-05-18 00:00:00
                )
        )
[1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [titre] => Second article
                    [contenu] => bbb
                    [created] => 2008-05-19 00:00:00
                )
        )
```

Ceci n'est pas nouveau : en fait, vous pouvez obtenir le même résultat sans le `behavior Containable` en faisant quelque chose comme :

```
$this->Post->recursive = -1;
$this->Post->find('all');
```

Le `behavior Containable` s'impose vraiment quand vous avez des associations complexes, et que vous voulez rogner le nombre d'information au même niveau. La propriété $recursive des models est utile si vous voulez éviter un niveau de recursivité entier, mais pas pour choisir ce que vous garder à chaque niveau. Regardons ensemble comment la methode `contain()` agit.

Le premier argument de la méthode accepte le nom, ou un tableau de noms, des models à garder lors du find. Si nous désirons aller chercher tous les posts et les tags annexes (sans aucune information de commentaire), nous devons essayer quelque chose comme

```
$this->Post->contain('Tag');
$this->Post->find('all');
```

Nous pouvons à nouveau utiliser la clé contain dans l'appel find() :

```
$this->Post->find('all', array('contain' => 'Tag'));
```

Sans le behavior Containable, nous finirions par utiliser la méthode `unbindModel()` du model, plusieurs fois si nous épluchons des models multiples. Le `behavior Containable` fourni un moyen plus propre pour accomplir cette même tâche.Contenant des associations plus profondes.

Des associations plus profondes

Le component Containable permet également d'aller un peu plus loin : vous pouvez filtrer les données des models associés . si vous regardez le résultats d'un appel find() classique, notez le champ "auteur" dans le model "Commentaire". Si vous êtes interéssés dans les posts par les noms et les commentaires des auteurs - et rien d'autre - vous devez faire quelque chose comme

```
$this->Post->contain('Comment.auteur');
$this->Post->find('all');

// ou..

$this->Post->find('all', array('contain' => 'Comment.auteur'));
```

ici , nous avons dit au behavior Containable de nous donner l'informations de post, et uniquement le champs auteur du model Commentaire associé. Le résultat du find ressemble à :

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [titre] => Premier article
                    [contenu] => aaa
                    [created] => 2008-05-18 00:00:00
                )
            [Commentaire] => Array
                (
```

```
                    [0] => Array
                        (
                            [auteur] => Daniel
                            [post_id] => 1
                        )
                    [1] => Array
                        (
                            [auteur] => Sam
                            [post_id] => 1
                        )
                )
        )
[1] => Array
        (...
```

Comme vous pouvez le voir, les tableaux de Comment ne contiennent uniquement que le champ auteur (avec le post_id qui est requit par CakePHP pour présenter le résultat)

Vous pouvez également filtrer les donneés associées à Comment en spécifiant une condition

```
$this->Post->contain('Comment.author = "Daniel"');
$this->Post->find('all');

//ou...

$this->Post->find('all', array('contain' => 'Comment.author = "Daniel"'));
```

Ceci nous donnes comme résultat les posts et commentaires dont daniel est l'auteur :

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => Premier article
                    [content] => aaa
                    [created] => 2008-05-18 00:00:00
                )
            [Comment] => Array
                (
                    [0] => Array
                        (
                            [id] => 1
                            [post_id] => 1
                            [author] => Daniel
                            [email] => dan@example.com
                            [website] => http://example.com
                            [comment] => Premier commentaire
                            [created] => 2008-05-18 00:00:00
                        )
                )
        )
```

Des filtre supplémentaires peuvent être utilisées en utilisant les options de recherche standard *find* :

```
$this->Post->find('all', array('contain' => array(
    'Comment' => array(
        'conditions' => array('Comment.author =' => "Daniel"),
        'order' => 'Comment.created DESC'
    )
)));
```

Voici un exemple d'utilisation de `ContainableBehavior` quand vous avez des relations profondes et complexes entre les models.

Examinons les associations des models suivants :

```
User->Profile
User->Account->AccountSummary
User->Post->PostAttachment->PostAttachmentHistory->HistoryNotes
User->Post->Tag
```

Voici comment nous recupérons les associations ci-dessus avec le behavior Containable

```
$this->User->find('all', array(
    'contain' => array(
        'Profile',
        'Account' => array(
            'AccountSummary'
        ),
        'Post' => array(
            'PostAttachment' => array(
                'fields' => array('id', 'name'),
                'PostAttachmentHistory' => array(
                    'HistoryNotes' => array(
                        'fields' => array('id', 'note')
                    )
                )
            ),
            'Tag' => array(
                'conditions' => array('Tag.name LIKE' => '%happy%')
            )
        )
    )
));
```

Gardez à l'esprit que la clé 'contain' n'est utilisée qu'une seule fois dans le model principal, vous n'avez pas besoin d'utiliser 'contain' a nouveau dans les models liés.

Note : En utilisant les options 'fields' et 'contain' - faîtes attention d'inclure toutes les clés étrangères que votre requête requiert directement ou indirectement. Notez également que c'est parce que le behavior Containable doit être attaché à tous les models utilisés dans le contenu, que vous devez l'attacher à votre AppModel.

Les options du Behavior Containable

Le `Behavior Containable` a plusieurs options qui peuvent être définies quand le behavior est attaché à un model. Ces paramètres vous permettent d'affiner le behavior de Containable et de travailler plus facilement avec les autres behaviors.

- **recursive** (boolean, optional), définir à true pour permettre au behavior Containable, de déterminer automatiquement le niveau de récursivité nécessaire pour récupérer les models spécifiés et de paramétrer la récursivité du model à ce niveau. Le définir à false désactive cette fonctionnalité. La valeur par défaut est `true`.
- **notices** (boolean, optional), émet des alertes E_NOTICES pour les liaisons référencées dans un appel containable et qui ne sont pas valides. La valeur par défaut est true.
- **autoFields** (boolean, optional), ajout automatique des champs nécessaires pour récupérer les liaisons requêtées. La valeur par défaut est `true`.

Vous pouvez changer les paramètres du Behavior Containable à l'exécution, en ré-attachant le behavior comme vu au chapitre *Behaviors (Comportements)* (Utiliser les Behaviors).

Le behavior Containable peut quelque fois causer des problèmes avec d'autres behaviors ou des requêtes qui utilisent des fonctions d'agrégations et/ou des clauses GROUP BY. Si vous obtenez des erreurs SQL invalides à cause du mélange de champs agrégés et non-agrégés, essayer de désactiver le paramètre `autoFields` :

```
$this->Post->Behaviors->load('Containable', array('autoFields' => false));
```

Utilisation du behavior Containable avec la pagination En incluant le paramètre 'contain' dans la propriété `$paginate`, la pagination sera appliqué à la fois au find('count') et au find('all') dans le model.

Voir la section *Utilisation de Containable* pour plus de détails.

Voici un exemple pour limiter les associations en paginant :

```
$this->paginate['User'] = array(
    'contain' => array('Profile', 'Account'),
    'order' => 'User.username'
);

$users = $this->paginate('User');
```

Le Behavior Translate

class `TranslateBehavior`

Le behavior Translate est en fait assez simple à paramétrer et à faire fonctionner out of the box, le tout avec très peu de configuration. Dans cette section, vous apprendrez comment ajouter et configurer ce behavior, pour l'utiliser dans n'importe quel model.

Si vous utilisez le behavior Translate en parallèle de Containable, assurez-vous de définir la clé 'fields' pour vos requêtes. Sinon, vous pourriez vous retrouver avec des fragments SQL générés invalides.

Initialisation des tables de la Base de donnée i18n

Vous pouvez soit utiliser la console CakePHP, soit les créer manuellement. Il est recommandé d'utiliser la console pour cela, parce qu'il pourrait arriver que le layout change dans les futures versions de CakePHP. En restant fidèle à la console, cela garantira que vous ayez le bon layout :

```
./cake i18n
```

Sélectionner''[I]'', ce qui lancera le script d'initialisation de la base de données i18n. Il vous sera demandé si vous voulez supprimer toute table existante et si vous voulez en créer une. Répondez par oui si vous êtes certain qu'il n'y a pas encore une table i18n et répondez encore par oui pour créer la table.

Attacher le Behavior Translate à vos Models

Ajoutez-le à votre model en utilisant la propriété `$actsAs` comme dans l'exemple suivant. :

```
class Post extends AppModel {
    public $actsAs = array(
        'Translate'
    );
}
```

Ceci ne produira encore rien, parce qu'il faut un couple d'options avant que cela ne commence à fonctionner. Vous devez définir, quels champs du model courant devront être détectés dans la table de traduction que nous avons créée précédemment.

Définir les Champs

Vous pouvez définir les champs en étendant simplement la valeur `'Translate'` avec un autre tableau, comme

```
class Post extends AppModel {
    public $actsAs = array(
        'Translate' => array(
            'fieldOne', 'fieldTwo', 'and_so_on'
        )
    );
}
```

Après avoir fait cela (par exemple, en précisant "title" comme l'un des champs), vous avez déjà terminé la configuration de base. Super ! D'après notre exemple courant, le model devrait maintenant ressembler à quelque chose comme cela

```
class Post extends AppModel {
    public $actsAs = array(
        'Translate' => array(
            'title'
        )
    );
}
```

Quand vous définissez vos champs à traduire dans le Behavior Translate, assurez-vous d'omettre les champs du schéma de model traduits. Si vous laissez les champs en place, il peut y avoir un problème de récupération de données avec les locales.

Conclusion

A partir de maintenant, chaque mise à jour/création d'un enregistrement fera que le Behavior Translate copiera la valeur de "title" dans la table de traduction (par défaut : i18n), avec la locale courante. Une "locale" est un identifiant d'une langue, pour ainsi dire.

Lire le contenu traduit

Par défaut, le TranslateBehavior va automatiquement récupèrer et ajouter les données basées sur la locale courante. La locale courante est lue à partir de `Configure::read('Config.language')` qui est assignée par la classe `L10n`. Vous pouvez surcharger cette valeur par défaut à la volée en utilisant `$Model->locale`.

Récupérer les champs traduits dans une locale spécifique En définissant `$Model->locale`, vous pouvez lire les traductions pour une locale spécifique :

```
// Lire les données de la locale espagnole.
$this->Post->locale = 'es';
$results = $this->Post->find('first', array(
    'conditions' => array('Post.id' => $id)
));
// $results va contenir la traduction espagnole.
```

Récupérer tous les enregistrements de traduction pour un champ Si vous voulez avoir tous les enregistrements de traduction attachés à l'enregistrement de model courant, vous étendez simplement le **tableau champ** dans votre paramétrage du behavior, comme montré ci-dessous. Vous êtes complètement libre de choisir le nommage. :

```
class Post extends AppModel {
    public $actsAs = array(
        'Translate' => array(
            'title' => 'titleTranslation'
        )
    );
}
```

Avec ce paramétrage, le résultat de votre `$this->Post->find()` devrait ressembler à quelque chose comme cela

```
Array
(
     [Post] => Array
         (
```

```
            [id] => 1
            [title] => Beispiel Eintrag
            [body] => lorem ipsum...
            [locale] => de_de
        )

    [titleTranslation] => Array
        (
            [0] => Array
                (
                    [id] => 1
                    [locale] => en_us
                    [model] => Post
                    [foreign_key] => 1
                    [field] => title
                    [content] => Example entry
                )

            [1] => Array
                (
                    [id] => 2
                    [locale] => de_de
                    [model] => Post
                    [foreign_key] => 1
                    [field] => title
                    [content] => Beispiel Eintrag
                )

        )
)
```

Note : L'enregistrement du model contient un champ *virtuel* appelé "locale". Il indique quelle locale est utilisée dans ce résultat.

Notez que seuls les champs du model que vous faîtes avec un 'find' seront traduits. Les Models attachés via les associations ne seront pas traduits parce que le déclenchement des callbacks sur les models associés n'est actuellement pas supporté.

Utiliser la méthode bindTranslation Vous pouvez aussi récupérer toutes les traductions seulement quand vous en avez besoin, en utilisant la méthode bindTranslation.

`TranslateBehavior::bindTranslation`(*$fields*, *$reset*)

`$fields` st un tableau associatif composé du champ et du nom de l'association, dans lequel la clé est le champ traduisible et la valeur est le nom fictif de l'association. :

```
$this->Post->bindTranslation(array('name' => 'titleTranslation'));
$this->Post->find('all', array('recursive' => 1)); // il est nécessaire d'avoir au moins u
```

Avec ce paramétrage, le résultat de votre find() devrait ressembler à quelque chose comme ceci

```
Array
(
     [Post] => Array
         (
             [id] => 1
             [title] => Beispiel Eintrag
             [body] => lorem ipsum...
             [locale] => de_de
         )

     [titleTranslation] => Array
         (
             [0] => Array
                 (
                     [id] => 1
                     [locale] => en_us
                     [model] => Post
                     [foreign_key] => 1
                     [field] => title
                     [content] => Example entry
                 )

             [1] => Array
                 (
                     [id] => 2
                     [locale] => de_de
                     [model] => Post
                     [foreign_key] => 1
                     [field] => title
                     [content] => Beispiel Eintrag
                 )

         )
)
```

Sauvegarder dans une autre Langue

Vous pouvez forcer le model qui utilise le TranslateBehavior à sauvegarder dans une autre langue que celle détectée.

Pour dire à un model dans quelle langue le contenu devra être sauvé, changez simplement la valeur de la propriété $locale du model, avant que vous ne sauvegardiez les données dans la base. Vous pouvez faire cela dans votre controller ou vous pouvez le définir directement dans le model.

Example A : Dans votre controller :

```
class PostsController extends AppController {

    public function add() {
        if (!empty($this->request->data)) {
            $this->Post->locale = 'de_de'; // we are going to save the german version
            $this->Post->create();
```

```
            if ($this->Post->save($this->request->data)) {
                $this->redirect(array('action' => 'index'));
            }
        }
    }
}
```

Exemple B : Dans votre model :

```
class Post extends AppModel {
    public $actsAs = array(
        'Translate' => array(
            'title'
        )
    );

    // Option 1) just define the property directly
    public $locale = 'en_us';

    // Option 2) create a simple method
    public function setLanguage($locale) {
        $this->locale = $locale;
    }
}
```

Traduction de Tables Multiples

Si vous attendez beaucoup d'entrée, vous vous demandez certainement comment gérer tout cela dans une base de données qui grossit rapidement. Il y a deux propriétés introduite dans le Behavior Translate qui permettent de spécifier quel model doit être relié au model qui contient les traductions.

Ceux-ci sont **$translateModel** et **$translateTable**.

Disons que nous voulons sauver nos traductions pour tous les posts dans la table "post_i18ns" au lieu de la valeur par défaut de la table "i18n". Pour faire cela vous avez besoin de paramétrer votre model comme cela

```
class Post extends AppModel {
    public $actsAs = array(
        'Translate' => array(
            'title'
        )
    );

    // Utilise un model différent (et table)
    public $translateModel = 'PostI18n';
}
```

Note : Il est important vous mettiez au pluriel la table. C'est maintenant un model habituel et il peut être traité en tant que tel avec les conventions qui en découlent. Le schéma de la table elle-même doit être identique à celui généré par la console CakePHP. Pour vous assurer qu'il s'intègre vous pourriez initialiser une table i18n vide au travers de la console et renommer la table après coup.

Créer le Model de Traduction Pour que cela fonctionne vous devez créer le fichier du model actuel dans le dossier des models. La raison est qu'il n'y a pas de propriété pour définir le displayField directement dans le model utilisant ce behavior.

Assurez vous de changer le `$displayField` en `'field'`.:

```
class PostI18n extends AppModel {
    public $displayField = 'field'; // important
}
// nom de fichier: PostI18n.php
```

C'est tout ce qu'il faut. Vous pouvez aussi ajouter toutes les propriétés des models comme $useTable. Mais pour une meilleure cohérence nous pouvons faire cela dans le model qui utilise ce model de traduction. C'est là que l'option `$translateTable` entre en jeu.

Modification de la Table Si vous voulez changer le nom de la table, il vous suffit simplement de définir $translateTable dans votre model, comme ceci

```
class Post extends AppModel {
    public $actsAs = array(
        'Translate' => array(
            'title'
        )
    );

    // Utilise un model différent
    public $translateModel = 'PostI18n';

    // Utilise une table différente pour translateModel
    public $translateTable = 'post_translations';
}
```

A noter que **vous ne pouvez pas utiliser $translateTable seule**. Si vous n'avez pas l'intention d'utiliser un Model de traduction `$translateModel` personnalisé, alors laissez cette propriété inchangée. La raison est qu'elle casserait votre configuration et vous afficherait un message "Missing Table" pour le model I18n par défaut, lequel est créé à l'exécution.

Tree

class `TreeBehavior`

C'est assez courant de vouloir stocker ses données sous une forme hiérarchique dans la table d'une base de données. Des exemples de tels besoins pourraient être des catégories avec un nombre illimité de sous-catégories, des données en relation avec un système de menu multi-niveaux ou une représentation littérale d'une hiérarchie, comme celle qui est utilisée pour stocker les objets de contrôle d'accès avec la logique ACL.

Pour de petits arbres de données et les cas où les données n'ont que quelques niveaux de profondeurs, c'est simple d'ajouter un champ parent_id à votre table et de l'utiliser pour savoir quel objet est le parent de quel autre. En natif avec CakePHP, il existe cependant un moyen puissant d'avoir les bénéfices de la logique MPTT *MPTT logic <http ://www.sitepoint.com/hierarchical-data-database-2/>*, sans avoir à connaître les détails de l'implémentation technique - à moins que ça ne vous intéresse ;).

Pré-requis

Pour utiliser le behavior en Arbre (TreeBehavior), votre table nécessite 3 champs tels que listés ci-dessous (tous sont des entiers) :
- parent - le nom du champ par défaut est parent_id, pour stocker l'id de l'objet parent.
- left - le nom du champ par défaut est lft, pour stocker la valeur lft de la ligne courante.
- right - le nom du champ par défaut est rght, pour stocker la valeur rght de la ligne courante.

Si vous êtes familier de la logique MPTT vous pouvez vous demander pourquoi un champ parent existe - parce qu'il est tout bonnement plus facile d'effectuer certaines tâches à l'usage, si un lien parent direct est stocké en base, comme rechercher les enfants directs.

Note : Le champ `parent` doit être capable d'avoir une valeur NULL ! Cela pourrait sembler fonctionner, si vous donnez juste une valeur parente de zéro aux éléments de premier niveau, mais le fait de réordonner l'arbre (et sans doute d'autres opérations) échouera.

Utilisation Basique

Le behavior Tree possède beaucoup de fonctionnalités, mais commençons avec un exemple simple. Créons la table suivante

```
CREATE TABLE categories (
    id INTEGER(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    parent_id INTEGER(10) DEFAULT NULL,
    lft INTEGER(10) DEFAULT NULL,
    rght INTEGER(10) DEFAULT NULL,
    name VARCHAR(255) DEFAULT '',
    PRIMARY KEY  (id)
);

INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(1, 'My Categorie
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(2, 'Fun', 1, 2,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(3, 'Sport', 2, 3
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(4, 'Surfing', 3,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(5, 'Extreme knit
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(6, 'Friends', 2,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(7, 'Gerald', 6,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(8, 'Gwendolyn',
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(9, 'Work', 1, 1
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(10, 'Reports', 9
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(11, 'Annual', 1
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(12, 'Status', 1
```

```
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(13, 'Trips', 9,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(14, 'National',
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(15, 'Internatio
```

Dans le but de vérifier que tout est défini correctement, nous pouvons créer une méthode de test et afficher les contenus de notre arbre de catégories, pour voir à quoi il ressemble. Avec un simple controller

```
class CategoriesController extends AppController {

    public function index() {
        $data = $this->Category->generateTreeList(null, null, null, '   ');
        debug($data); die;
    }
}
```

et une définition de model encore plus simple

```
// app/Model/Category.php
class Category extends AppModel {
    public $actsAs = array('Tree');
}
```

Nous pouvons vérifier à quoi ressemble les données de notre arbre de catégories, en visitant /categories. Vous devriez voir quelque chose comme :

- My Categories
  - Fun
    - Sport
      - Surfing
      - Extreme knitting
    - Friends
      - Gerald
      - Gwendolyn
  - Work
    - Reports
      - Annual
      - Status
    - Trips
      - National
      - International

Ajouter des données Dans la section précédente, nous avons utilisé des données existantes et nous avons vérifié qu'elles semblaient hiérarchiques avec la méthode `generateTreeList`. Toutefois vous devez ajouter vos données de la même manière que vous le feriez pour n'importe quel model. Par exemple

```
// pseudo controller code
$data['Category']['parent_id'] = 3;
$data['Category']['name'] = 'Skating';
$this->Category->save($data);
```

Lorsque vous utilisez le behavior en arbre il n'est pas nécessaire de faire plus que de définir l'id du parent (parent_id), le behavior tree prendra soin du reste. Si vous ne définissez pas l'id du parent (parent_id), Le behavior Tree additionnera vos nouveaux ajouts au sommet de l'arbre :

```
// pseudo code du controller
$data = array();
$data['Category']['name'] = 'Other People\'s Categories';
$this->Category->save($data);
```

Exécuter les extraits de code ci-dessus devrait modifier l'arbre comme suit :

- My Categories
  - Fun
    - Sport
      - Surfing
      - Extreme knitting
      - Skating **New**
    - Friends
      - Gerald
      - Gwendolyn
  - Work
    - Reports
      - Annual
      - Status
    - Trips
      - National
      - International
- Other People's Categories **New**

Modification des données La modification des données est aussi transparente que l'addition des données. Si vous modifiez quelque chose, mais ne changez pas le champ de l'id du parent (parent_id) - la structure de vos données restera inchangée. Par exemple

```
// pseudo controller code
$this->Category->id = 5; // id of Extreme knitting
$this->Category->save(array('name' => 'Extreme fishing'));
```

Le code ci-dessus n'affecterait pas le champ de l'id du parent (parent_id) - même si l'id du parent (parent_id) est inclue dans les données passées à sauvegarder si les données ne changent pas, pas plus que la structure de données. Donc l'arbre de données devrait maintenant ressembler à :

- My Categories
- Fun

- Sport
  - Surfing
  - Extreme fishing **Updated**
  - Skating
- Friends
  - Gerald
  - Gwendolyn

- Work
- Reports
  - Annual
  - Status
- Trips
  - National
  - International
- Other People's Categories

Déplacer les données autour de votre arbre est aussi une affaire simple. Supposons que Extreme fishing n'appartienne pas à Sport, mais devrait se trouver plutôt sous "D'autres catégories de gens". Avec le code suivant

```
// pseudo code du controller
$this->Category->id = 5; // id de Extreme fishing
$newParentId = $this->Category->field('id', array('name' => 'Other People\'s Categories'));
$this->Category->save(array('parent_id' => $newParentId));
```

Comme on pouvait s'y attendre, la structure serait modifiée comme suit :

- My Categories
- Fun
  - Sport
    - Surfing
    - Skating
  - Friends
    - Gerald
    - Gwendolyn
- Work
  - Reports
    - Annual
    - Status
  - Trips
    - National
    - International
- Other People's Categories
- Extreme fishing **déplacé**

Suppression des données Le behavior Tree fournit un certain nombre de façons de gérer la suppression des données. Pour commencer par le plus simple exemple, disons que la catégorie des rapports n'est plus utile. Pour l'enlever * et tous les enfants qu'il peut avoir * il suffit d'appeler et supprimer comme vous le feriez pour n'importe quel model. Par exemple, avec le code suivant

```
// pseudo code du controller
$this->Category->id = 10;
$this->Category->delete();
```

L'arbre des Catégories serait modifié comme suit :

- My Categories
- Fun
  - Sport
    - Surfing
    - Skating
  - Friends
    - Gerald
    - Gwendolyn
- Work
  - Trips
    - National
    - International
- Other People's Categories
- Extreme fishing

Interroger et utiliser vos données Utiliser et manipuler des données hiérarchisées peut s'avérer assez difficile. C'est pourquoi le behavior tree met à votre disposition quelques méthodes de permutations en plus des méthodes find de bases.

Note : La plupart des méthodes de tree se basent et renvoient des données triées en fonction du champ `lft`. Si vous appelez `find()` sans trier en fonction de `lft`, ou si vous faîtes une demande de tri sur un tree, vous risquez d'obtenir des résultats inattendus.

---

class `TreeBehavior`

`children`(*$id = null*, *$direct = false*, *$fields = null*, *$order = null*, *$limit = null*, *$page = 1*, *$recursive = null*)

Paramètres

- **$id** – L'id de l'enregistrement à rechercher.
- **$direct** – Defini à true pour ne retourner que les descendants directs.
- **$fields** – Un simple champ texte ou un tableau de champs à inclure dans le retour.
- **$order** – Chaîne SQL des conditions ORDER BY.
- **$limit** – SQL LIMIT déclaration.
- **$page** – pour accéder aux resultats paginés.
- **$recursive** – Nombre du niveau de profondeur pour la recursivité des models associés.

La méthode `children` prend la clé primaire (l'id d'une ligne) et retourne l'enfant (children), par défaut dans l'ordre d'apparition dans l'arbre. Le second paramètre optionnel definit si il faut ou non retourner les enfants directs. En utilisant l'exemple des données de la section précédente :

```
$allChildren = $this->Category->children(1); // un tableau plat à 11 éléments
        // -- ou --
$this->Category->id = 1;
$allChildren = $this->Category->children(); // un tableau plat à 11 éléments
```

```
// Ne retourne que les enfants directs
$directChildren = $this->Category->children(1, true); // un tableau plat avec 2 élémen
```

Note : Si vous voulez un tableau recursif utilisez `find('threaded')`

childCount (*$id = null*, *$direct = false*)

Comme avec la méthode `children`, `childCount` prend la valeur de la clé primaire (l'id) d'une ligne et retourne combien d'enfant elle contient.

Le second paramètre optionnel definit si il faut ou non compter les enfants directs. En reprenant l'exemple ci dessus

```
$totalChildren = $this->Category->childCount(1); // retournera 11
// -- ou --
$this->Category->id = 1;
$directChildren = $this->Category->childCount(); //retournera 11

// Seulement les comptes des descendants directs de cette category
$numChildren = $this->Category->childCount(1, true); // retournera 2
```

generateTreeList (*$conditions=null*, *$keyPath=null*, *$valuePath=null*, *$spacer= '_'*, *$recursive=null*)

Paramètres

- **$conditions** – Utilise les mêmes conditions qu'un find().
- **$keyPath** – Chemin du champ à utiliser pour la clé.
- **$valuePath** – Chemin du champ à utiliser pour le label.
- **$spacer** – La chaîne à utiliser devant chaque élément pour indiquer la profondeur.
- **$recursive** – Le nombre de niveaux de profondeur pour rechercher les enregistrements associés.

Cette méthode retourne des données similaires à :ref : *model-find-list*, avec un préfixe en retrait pour montrer la structure de vos données. Voici un exemple de ce à quoi vous attendre comme retour avec cette méthode

```
$treelist = $this->Category->generateTreeList();
```

Sortie :

```
array(
    [1] =>  "My Categories",
    [2] =>  "_Fun",
    [3] =>  "__Sport",
    [4] =>  "___Surfing",
    [16] => "___Skating",
    [6] =>  "__Friends",
    [7] =>  "___Gerald",
    [8] =>  "___Gwendolyn",
    [9] =>  "_Work",
    [13] => "__Trips",
    [14] => "___National",
    [15] => "___International",
    [17] => "Other People's Categories",
```

```
        [5] =>  "_Extreme fishing"
    )
```

getParentNode()

Cette fonction comme son nom l'indique, donne en retour le noeud parent d'un nœud, ou *false* si le noeud n'a pas de parent (c'est le nœud racine). Par exemple

```
$parent = $this->Category->getParentNode(2); //<- id de fun
// $parent contient toutes les catégories
```

getPath(*$id = null*, *$fields = null*, *$recursive = null*)

Le 'path' (chemin) quand vous vous référez à des données hiérarchiques, c'est comment retrouver où vous êtes depuis le sommet. Par exemple le path (chemin) de la catégorie "International" est :
–My Categories
–...
–Work
–Trips
–...
–International

En utilisant l'id de "international", getPath retournera chacun des parents rencontrés (depuis le haut) :

```
$parents = $this->Category->getPath(15);
```

```
// contenu de $parents
array(
    [0] =>  array('Category' => array('id' => 1, 'name' => 'My Categories', ..)),
    [1] =>  array('Category' => array('id' => 9, 'name' => 'Work', ..)),
    [2] =>  array('Category' => array('id' => 13, 'name' => 'Trips', ..)),
    [3] =>  array('Category' => array('id' => 15, 'name' => 'International', ..)),
)
```

Utilisation avancée

Le behavior Tree ne fonctionne pas uniquement en tâche de fond, il y a un certain nombre de méthodes spécifiques dans le behavior Tree pour répondre a vos besoins de données hierarchiques, et des problèmes inattendus qui pourraient survenir durant le processus.

TreeBehavior::**moveDown**()

Utilisé pour déplacer un seul nœud dans l'arbre. Vous devez fournir l' ID de l'élément à déplacer et un nombre positif de combien de positions le noeud devrait être déplacé vers le bas. Tous les nœuds enfants pour le noeud spécifié seront également déplacés.

Voici l'exemple d'une action d'un controller (dans un controller nommé Category) qui déplace un noeud spécifié vers le bas de l'arbre :

```
public function movedown($id = null, $delta = null) {
    $this->Category->id = $id;
    if (!$this->Category->exists()) {
       throw new NotFoundException(__('Invalid category'));
    }
```

```
        if ($delta > 0) {
            $this->Category->moveDown($this->Category->id, abs($delta));
        } else {
            $this->Session->setFlash('Please provide the number of positions the field sho
        }

        $this->redirect(array('action' => 'index'));
    }
```

Par exemple, si vous souhaitez déplacer le "Sport" (id de 3) d'une catégorie vers le bas, vous devriez requêter : /categories/movedown/3/1.

`TreeBehavior::moveUp()`

Utilisé pour déplacer un seul nœud de l'arbre. Vous devez fournir l'ID de l'élément à déplacer et un nombre positif de combien de positions le noeud devrait être déplacé vers le haut. Tous les nœuds enfants seront également déplacés.

Voici un exemple d'un controller action (dans un controller categories) déplacant un noeud plus haut dans un arbre :

```
    public function moveup($id = null, $delta = null) {
        $this->Category->id = $id;
        if (!$this->Category->exists()) {
           throw new NotFoundException(__('Invalid category'));
        }

        if ($delta > 0) {
            $this->Category->moveUp($this->Category->id, abs($delta));
        } else {
            $this->Session->setFlash('Please provide a number of positions the category sh
        }

        $this->redirect(array('action' => 'index'));
    }
```

Par exemple, si vous souhaitez déplacer la catégory "Gwendoline" (id de 8) plus haut d'une position vous devriez requêter : /categories/moveup/8/1. Maintenant l'ordre des Amis sera Gwendolyn, Gérald.

`TreeBehavior::removeFromTree($id = null, $delete = false)`

En utilisant cette méthode, un neud sera supprimée ou déplacée, tout en conservant son sous-arbre, qui sera apparenté à un niveau supérieur. Il offre plus de contrôle que : ref : *model-delete* qui, pour un model en utilisant le behavior tree supprimera le noeud spécifié et tous ses enfants.

Prenons l'arbre suivant au début :

- My Categories
  - Fun
    - Sport
      - Surfing
      - Extreme knitting
      - Skating

En executant le code suivant avec l'id de 'Sport' :

```
$this->Node->removeFromTree($id);
```

Le noeud Sport sera retiré du haut du noeud :

- My Categories
- Fun
  - Surfing
  - Extreme knitting
  - Skating
- Sport **Déplacé**

Cela démontre le behavior par défaut du `removeFromTree` de déplacement d'un noeud pour ne plus avoir de parent, et de re-parenter tous les enfants.

Si toutefois l'extrait de code suivant était utilisé avec l'id 'Sport' :

```
$this->Node->removeFromTree($id, true);
```

L'arbre deviendrait.

- My Categories
  - Fun
    - Surfing
    - Extreme knitting
    - Skating

Ceci démontre l'utilisation alternative de `removeFromTree`, les enfants ont été reparentés et 'Sport' a été effacé.

`TreeBehavior::`**`reorder`**(*array('id' => null, 'field' => $Model->displayField, 'order' => 'ASC', 'verify' => true)*)

Réordonne les nœuds (et nœuds enfants) de l'arbre en fonction du champ et de la direction spécifiée dans les paramètres. Cette méthode ne changera pas le parent d'un nœud. :

```
$model->reorder(array(
    'id' => ,     //id de l\'enregistrement à utiliser comme noeud haut pour réordonner
    'field' => , //champ à utiliser pour réordonner, par défaut: $Model->displayField
    'order' => , //direction de l\'ordonnement, par défaut: 'ASC'
    'verify' =>  //vérifier ou pas l'arbre avant de réordonner, par défaut: true
));
```

Note : Si vous avez sauvegardé vos données ou fait d'autres opérations sur le model, vous pouvez définir `$model->id = null` avant d'appeler `reorder`. Sinon, seuls les enfants du nœud actuel et ses enfants seront réordonnés.

Intégrité des données

En raison de la nature complexe auto-référentielle de ces structures de données comme les arbres et les listes chaînées, elles peuvent parfois se rompre par un appel négligent. Rassurez-vous, tout n'est pas perdu ! Le behavior Tree contient plusieurs fonctionnalités précédemment non-documentées destinées à se remettre de telles situations.

`TreeBehavior::`**`recover`**(*$mode = 'parent', $missingParentAction = null*)

Le paramètre `mode` est utilisé pour spécifier la source de l'info qui est correcte. La source opposée de données sera peuplée en fonction de cette source d'information. Ex : si le champ MPTT est corrompu ou vide, avec le `$mode 'parent'` la valeur du champ `parent_id` sera utilisée pour peupler les champs gauche et droite.

Le paramètre `missingParentAction` s'applique uniquement aux "parent" mode et détermine ce qu'il faut faire si le champ parent contient un identifiant qui n'est pas présent.

Options `$mode` permises :
- `'parent'` - utilise l'actuel''parent_id''pour mettre à jour les champs `lft` and `rght`.
- `'tree'` - utilise les champs actuels `lft``et``rght``pour mettre à jour le champ ``parent_id`

Les options de `missingParentActions` autorisées durant l'utilisation de `mode='parent'` :
- `null` - ne fait rien et continue
- `'return'` - ne fait rien et fait un return
- `'delete'` - efface le noeud
- `int` - definit parent_id à cet id

Exemple :

```
// Reconstruit tous les champs gauche et droit en se basant sur parent_id
$this->Category->recover();
// ou
$this->Category->recover('parent');

// Reconstruit tous les parent_id en se basant sur les champs lft et rght
$this->Category->recover('tree');
```

`TreeBehavior::`**`reorder`**(*$options = array()*)

Réordonne les nœuds (et nœuds enfants) de l'arbre en fonction du champ et de la direction spécifiés dans les paramètres. Cette méthode ne change pas le parent d'un nœud.

La réorganisation affecte tous les nœuds dans l'arborescence par défaut, mais les options suivantes peuvent influer sur le processus :
- `'id'` - ne réordonne que les noeuds sous ce noeud.
- `'field'` - champ à utiliser pour le tri, par défaut le `displayField` du model.
- `'order'` - `'ASC'` pour tri ascendant, `'DESC'` pour tri descendant.
- `'verify'` - avec ou sans vérification avant tri.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et les clés suivantes par défaut, toutes sont facultatives :

```
array(
    'id' => null,
    'field' => $model->displayField,
    'order' => 'ASC',
    'verify' => true
)
```

`TreeBehavior::`**`verify`**()

Retourne `True` si l'arbre est valide sinon un tableau d'erreurs, avec des champs pour le type, l'index, et le message d'erreur.

Chaque enregistrement dans le tableau de sortie est un tableau de la forme (type, id,message)
- `type` est soit `'index'` ou `'node'`

– `'id'` est l'id du noeud erroné.
– `'message'` dépend de l'erreur rencontrée

Exemple d'utilisation :

```
$this->Category->verify();
```

Exemple de sortie :

```
Array
(
    [0] => Array
        (
            [0] => node
            [1] => 3
            [2] => left and right values identical
        )
    [1] => Array
        (
            [0] => node
            [1] => 2
            [2] => The parent node 999 doesn't exist
        )
    [10] => Array
        (
            [0] => index
            [1] => 123
            [2] => missing
        )
    [99] => Array
        (
            [0] => node
            [1] => 163
            [2] => left greater than right
        )

)
```

Components (Composants)

CakePHP a une sélection de components pour aider à s'occuper de tâches basiques dans vos controllers. Regardez la section sur *Components (Composants)* pour savoir comment configurer et utiliser les components.

Components (Composants)

Liste de contrôle d'accès (ACL)

class `AclComponent` (*ComponentCollection $collection*, *array $settings = array()*)

La fonctionnalité de listes de contrôle d'accès (Access Control List, ACL) de CakePHP est l'une des plus souvent discutée, probablement parce qu'elle est la plus recherchée, mais aussi parce qu'elle peut-être la plus déroutante. Si vous recherchez une bonne façon de débuter avec les ACLs en général, lisez ce qui suit.

Soyez courageux et persévérant avec ce sujet, même si au départ cela paraît difficile. Une fois que vous aurez pris le coup, ce sera un outil extrêmement puissant, à garder sous la main quand vous développez votre application.

Comprendre le fonctionnement des ACL

Les choses importantes requièrent un contrôle d'accès. Les listes de contrôles d'accès sont une façon de gérer les permissions d'une application d'une manière très précise et pourtant facilement maintenable et manipulable.

Les listes de contrôles d'accès, ou ACL (Access Control Lists), manipulent deux choses principales : les choses qui veulent accéder à des trucs et celles qui sont recherchées. Dans le jargon ACL, les choses qui veulent accéder à des trucs (le plus souvent les utilisateurs) sont appelées access request objects (objets requête d'accès) ou AROs. Les choses du système qui sont recherchées (le plus souvent les actions ou les données) sont appelées access control objects (objets contrôle d'accès) ou ACOs. Les entités sont appelées "objets", parce que parfois, l'objet demandé n'est pas une personne - des fois, vous pourriez vouloir limiter l'accès à certains controllers de Cake qui doivent initier leur logique dans d'autres parties de votre application. Les ACOs pourraient être n'importe quoi que vous voudriez contrôler, d'une action de controller à un service Web, en passant par une case de l'agenda en ligne de votre Mamy.

Rappel :
- ACO - Objet Contrôle d'Accès - Quelque chose qui est recherché
- ARO - Objet Requête d'Accès - Quelque chose qui veut quelque chose

Généralement, les ACL sont utilisées pour décider quand un ARO peut obtenir l'accès à un ACO.

Afin de vous aider à comprendre comment toutes les choses travaillent ensemble, utilisons un exemple semi-fonctionnel. Imaginons un moment, un ordinateur utilisé par un célèbre groupe d'aventuriers tirés du roman fantastique le *Seigneur des Anneaux*. Le chef du groupe, Gandalf, veut gérer les biens du groupe, tout en maintenant un bon niveau de confidentialité et de sécurité entre les autres membres de l'équipe. La première chose dont il a besoin est de créer une liste d'AROs qui comprend :
- Gandalf
- Aragorn
- Bilbo
- Frodo
- Gollum
- Legolas
- Gimli
- Pippin
- Merry

Note : Comprenez que l'ACL n'est pas la même chose que l'authentification. L'ACL est ce qui vient après qu'un utilisateur ait été authentifié. Par contre, les deux sont habituellement utilisés de paire, il est important de faire la distinction entre savoir qui est quelqu'un (authentification) et savoir ce qu'il peut faire (ACL).

La chose suivante que Gandalf doit faire, c'est de créer une liste initiale des choses, ou ACOs, que le système va contrôler. Sa liste devrait ressembler à quelque chose comme ça :
- Les armes
- L'Anneau
- Le porc salé
- La diplomatie
- La bière

Traditionnellement, les systèmes étaient gérés en utilisant une sorte de matrice, qui présentait un ensemble basique d'utilisateurs et de permissions en relation avec les objets. Si ces informations étaient stockées dans un tableau, il ressemblerait à ça :

x	Les armes	L'anneau	Le porc salé	La diplomacie	La bière
Gandalf			Autorisé	Autorisé	Autorisé
Aragorn	Autorisé		Autorisé	Autorisé	Autorisé
Bilbo					Autorisé
Frodo		Autorisé			Autorisé
Gollum			Autorisé		
Legolas	Autorisé		Autorisé	Autorisé	Autorisé
Gimli	Autorisé		Autorisé		
Pippin				Autorisé	Autorisé
Merry				Autorisé	Autorisé

A première vue, il semble que ce système pourrait très bien fonctionner. Les affectations peuvent être mises en place à des fin de sécurité (seul Frodo peut accéder à l'Anneau) et pour éviter les accidents (en gardant les hobbits à distance du porc salé et des armes). Cela paraît suffisamment complet et assez facile à lire, n'est-ce pas ?

Pour un petit système comme celui-ci, peut-être qu'une configuration en matrice pourrait fonctionner. Mais pour un système évolutif ou un système avec un fort pourcentage de ressources (ACOs) et d'utilisateurs (AROs), un tableau peut devenir plus lourd que rapide.

Imaginez une tentative de contrôler l'accès à des centaines de camps militaires et de gérer cela par unité. Un autre inconvénient des matrices est que vous ne pouvez par vraiment regrouper logiquement des sections d'utilisateurs ou faire des changements de permissions en cascade, pour des groupes d'utilisateurs basés sur ces regroupements logiques. Par exemple, il serait certainement plus chouette d'autoriser automatiquement les hobbits à accéder à la bière et au porc une fois que le combat est fini : faire ça sur une base d'utilisateurs gérés individuellement pourrait être fastidieux et source d'erreur. Faire des changements de permissions en cascade pour tous les "hobbits" serait plus facile.

Les ACL sont très souvent implémentés dans une structure en arbre. Il y a généralement un arbre d'AROs et un arbre d'ACOs. En organisant vos objets en arbres, les permissions peuvent toujours être distribuées d'une façon granulaire, tout en maintenant encore une bonne cohérence de l'ensemble. En chef raisonnable qu'il est, Gandalf choisit d'utiliser l'ACL dans son nouveau système et d'organiser ses objets de la manière suivante :
- La Communauté de l'Anneau™
  - Les Guerriers
    - Aragorn
    - Legolas
    - Gimli
  - Les Magiciens

    - Gandalf
  - Les Hobbits
    - Frodo
    - Bilbo
    - Merry
    - Pippin
  - Les Visiteurs
    - Gollum

L'utilisation d'une structure en arbre pour les AROs permet à Gandalf, de définir en une fois des autorisations qui s'appliquent à un groupe entier d'utilisateurs. Ainsi, en utilisant notre arbre ARO, Gandalf peut ajouter, après coup, quelques permissions de groupe :

- La Communauté de l'Anneau (**Refuser** : tout)
  - Guerriers (**Autoriser** :Armes, Bière, Rations pour les Elfes, Porc salé)
    - Aragorn
    - Legolas
    - Gimli
  - Magiciens (**Autoriser** : Porc salé, Diplomatie, Bière)
    - Gandalf
  - Hobbits (**Autoriser** : Bière)
    - Frodo
    - Bilbo
    - Merry
    - Pippin
  - Visiteurs (**Autoriser** : Porc salé)
    - Gollum

Si nous voulions utiliser les ACL pour voir si Pippin était autorisé à accéder à la bière, nous devrions d'abord récupérer son chemin dans l'arbre, lequel est Communauté->Hobbits->Pippin. Ensuite nous verrions les différentes permissions qui résident à chacun de ces points et nous utiliserions la plus spécifique des permissions reliant Pippin et la bière.

ARO Node	Permission Info	Result
Fellowship of the Ring	Deny all	Denying access to ale.
Hobbits	Allow 'ale'	Allowing access to ale !
Pippin	–	Still allowing ale !

Note : Puisque le nœud "Pippin" dans l'arbre d'ACL ne refuse pas spécifiquement l'accès à l'ACO bière, le résultat final est que nous donnons l'accès à cet ACO.

L'arbre nous permet aussi de faire des ajustements plus fins pour un meilleur contrôle granulaire, tout en conservant encore la capacité de faire de grands changements pour les groupes d'AROs :

- Communauté de l'Anneau (**Refuser** : tout)
  - Guerriers (**Autoriser** : Armes, Bière, Rations pour les Elfes, Porc salé)
    - Aragorn (Autoriser : Diplomatie)
    - Legolas
    - Gimli
  - Magiciens (**Autoriser** : Porc salé, Diplomatie, Bière)
    - Gandalf
  - Hobbits (**Autoriser** : Bière)

  - Frodo (Autoriser : Anneau)
  - Bilbo
  - Merry (Refuser : Bière)
  - Pippin (Autoriser : Diplomatie)
- Visiteurs (**Autoriser** : Porc salé)
  - Gollum

Cette approche nous donne plus de possibilités pour faire des changements de permissions de grande ampleur, mais aussi des ajustements plus précis. Cela nous permet de dire que tous les hobbits peuvent accéder à la bière, avec une exception — Merry. Pour voir si Merry peut accéder à la bière, nous aurions trouvé son chemin dans l'arbre : Communauté->Hobbits->Merry et appliqué notre principe, en gardant une trace des permissions liées à la bière :

Nœud de l'ARO Information sur la permission Résultat Communauté de l'Anneau Refuse tout Refuser l'accès à la bière. Hobbits Autorise la bière Autoriser l'accès à la bière ! Merry Refuse la bière Refuser la bière

Nœud de l'ARO	Information sur la permission	Résultat
Communauté de l'Anneau	Refuse tout	Refuser l'accès à la bière.
Hobbits	Autorise la bière	Autoriser l'accès à la bière !
Merry	Refuse la bière	Refuser la bière.

Définir les permissions : ACL de Cake basées sur des fichiers INI

La première implémentation d'ACL sur Cake était basée sur des fichiers INI stockés dans l'installation de Cake. Bien qu'elle soit stable et pratique, nous recommandons d'utiliser plutôt les solutions d'ACL basées sur les bases de données, surtout pour leur capacité à créer de nouveaux ACOs et AROs à la volée. Nous recommandons son utilisation dans de simples applications - et spécialement pour ceux qui ont une raison plus ou moins particulière de ne pas vouloir utiliser une base de données.

Par défaut, les ACL de CakePHP sont gérés par les bases de données. Pour activer les ACL basés sur les fichiers INI, vous devez dire à CakePHP quel système vous utilisez en mettant à jour les lignes suivantes dans app/config/core.php

```
// Changer ces lignes :
Configure::write('Acl.classname', 'DbAcl');
Configure::write('Acl.database', 'default');
// Pour qu'elles ressemblent à çà :
Configure::write('Acl.classname', 'IniAcl');
//Configure::write('Acl.database', 'default');
```

Les permissions des ARO/ACO sont spécifiées dans /app/config/acl.ini.php. L'idée de base est que les AROs sont spécifiés dans une section INI qui a trois propriétés : groups, allow et deny.

> groups : nom du groupe dont l'ARO est membre. allow : nom des ACOs auxquels l'ARO a accès. deny : nom des ACOs auxquels l'ARO ne devrait pas avoir accès.

Les ACOs sont spécifiés dans des sections INI qui incluent seulement les propriétés allow et deny.

Par exemple, voyons à quoi la structure ARO de la Communauté que nous avions façonnée pourrait ressembler dans une syntaxe INI :

```
;-------------------------------------
    ; AROs
;-------------------------------------
[aragorn]
groups = guerriers
allow = diplomatie

[legolas]
groups = guerriers

[gimli]
groups = guerriers

[gandalf]
groups = magiciens

[frodo]
groups = hobbits
allow = anneau

[bilbo]
groups = hobbits

[merry]
groups = hobbits
deny = ale

[pippin]
groups = hobbits

[gollum]
groups = visiteurs

;-------------------------------------
; ARO Groups
;-------------------------------------
[guerriers]
allow = armes, biere, porc_sale

[magiciens]
allow = porc_sale, diplomatie, biere

[hobbits]
allow = biere

[visiteurs]
allow = porc_sale
```

Maintenant que vous avez défini vos permissions, vous pouvez passer à la section sur la *vérification des permissions* en utilisant le component ACL.

Définir les permissions : ACL de Cake via une base de données

Maintenant que nous avons vu les permissions ACL basées sur les fichiers INI, voyons les ACL via une base de données (les plus communément utilisées).

Pour commencer L'implémentation par défaut des permissions ACL est propulsé par les bases de données. La base de données Cake pour les ACL est composé d'un ensemble de models du cœur et d'une application en mode console qui sont créés lors de votre installation de Cake. Les models sont utilisés par Cake pour interagir avec votre base de données, afin de stocker et de retrouver les nœuds sous forme d'arbre. L'application en mode console est utilisée pour initialiser votre base de données et interagir avec vos arbres d'ACO et d'ARO.

Pour commencer, vous devrez d'abord être sûr que votre `/app/config/database.php` est présent et correctement configuré. Voir la section 4.1 pour plus d'information sur la configuration d'une base de données.

Une fois que vous l'avez fait, utilisez la console de CakePHP pour créer vos tables d'ACL :

```
$ cake schema create DbAcl
```

Lancer cette commande va supprimer et recréer les tables nécessaires au stockage des informations des ACO et des ARO sous forme d'arbre. La sortie console devrait ressembler à quelque chose comme ça :

```
---------------------------------------------------------------
Cake Schema Shell
---------------------------------------------------------------

The following tables will be dropped.
acos
aros
aros_acos

Are you sure you want to drop the tables? (y/n)
[n] > y
Dropping tables.
acos updated.
aros updated.
aros_acos updated.

The following tables will be created.
acos
aros
aros_acos

Are you sure you want to create the tables? (y/n)
[y] > y
Creating tables.
acos updated.
aros updated.
aros_acos updated.
End create.
```

Note : Ceci remplace une commande désuète et dépréciée, "initdb".

Vous pouvez aussi vous servir du fichier SQL que vous trouverez dans app/config/sql/db_acl.sql, mais ce sera moins sympa.

Quand ce sera fini, vous devriez avoir trois nouvelles tables dans votre système de base de données : acos, aros et aros_acos (la table de jointure pour créer les permissions entre les deux arbres).

Note : Si vous êtes curieux de connaitre la façon dont Cake stocke l'information de l'arbre dans ces tables, étudiez l'arbre transversal sur la base de données modifiée. Le component ACL utilise le comportement en arbre de CakePHP pour gérer les héritages d'arbres. Les fichiers de model de classe pour ACL sont compilés dans un seul fichier db_acl.php. db_acl.php (http ://api.cakephp.org/file/cake/libs/model/db_acl.php).

Maintenant que nous avons tout configuré, attelons-nous à la création de quelques arbres ARO et ACO.

Créer des Objet Contrôle d'Accès (ACOs) et des Objet Requête d'Accès (AROs) Pour la création de nouveaux objets (ACOs et AROs), il y a deux principales façons de nommer et d'accéder aux noeuds. La première méthode est de lier un objet ACL directement à un enregistrement dans votre base de données en spécifiant le nom du model et la clé étrangère. La seconde méthode peut être utilisée quand un objet n'est pas en relation directe avec un enregistrement de votre base de données - vous pouvez fournir un alias textuel pour l'objet.

Note : Généralement, quand vous créez un groupe ou un objet de niveau supérieur, nous recommandons d'utiliser un alias. Si vous gérez l'accès à un enregistrement ou à un article particulier de la base de données, nous recommandons d'utiliser la méthode du model/clé étrangère.

Vous voulez créer de nouveaux objets ACL en utilisant le model ACL du coeur de CakePHP. Pour ce faire, il y a un nombre de champs que vous aurez à utiliser pour enregistrer les données : `model`, `foreign_key`, `alias`, et `parent_id`.

Les champs `model` et `foreign_key` pour un objet ACL vous permettent de créer un lien entre les objets qui correspondent à l'enregistrement du model (s'il en est). Par exemple, un certain nombre d'AROs correspondraient aux enregistrements User de la base de données. Il faut configurer la `foreign_key` pour que l'ID du User vous permette de lier les informations de l'ARO et de l'User avec un seul appel find() au model User avec la bonne association. Réciproquement, si vous voulez gérer les opérations d'édition sur un article spécifique d'un blog ou d'une liste de recette, vous devez choisir de lier un ACO à cet enregistrement spécifique du model.

L'`alias` d'un objet ACL est un simple label lisible pour un humain que vous pouvez utiliser pour identifier un objet ACL qui n'est pas en relation directe avec un enregistrement d'un model. Les alias sont couramment utilisés pour nommer les groupes d'utilisateurs ou les collections d'ACOs.

Le `parent_id` d'un objet ACL vous permet de remplir la structure de l'arbre. Il fournit l'ID du noeud parent dans l'arbre pour créer un nouvel enfant.

Avant que vous ne puissiez créer de nouveaux objets ACL, nous devront charger leurs classes respectives. La façon la plus facile de le faire est d'inclure les components ACL de Cake dans le tableau $composents

du controller :

```
public $components = array('Acl');
```

Quand ce sera fait, nous verrons quelques exemples de création de ces objets. Le code suivant pourrait être placé quelque part dans l'action d'un controller :

Note : Tant que les exemples que nous voyons ici nous montrent la création d'ARO, les mêmes techniques pourront être utilisées pour la création d'un arbre d'ACO.

Pour rester dans notre configuration de Communauté, nous allons d'abord créer nos groupes d'ARO. Comme nos groupes n'ont pas réellement d'enregistrements spécifiques qui leurs soient reliés, nous allons utiliser les alias pour créer ces objets ACL. Ce que nous faisons ici est en perspective d'une action du controller mais pourrait être fait ailleurs. Ce que nous allons aborder ici est un peu une approche artificielle, mais vous devriez trouver ces techniques plus confortables à utiliser pour créer des AROs et des ACOs à la volée.

Ceci ne devrait rien avoir de radicalement nouveau - nous sommes juste en train d'utiliser les models pour enregistrer les données comme nous le faisons toujours :

```
public function touteslesActions() {
    $aro =& $this->Acl->Aro;
    //Ici ce sont toutes les informations sur le tableau de notre groupe que nous
    //pouvons itérer comme ceci
    $groups = array(
        0 => array(
        'alias' => 'guerriers'
        ),
        1 => array(
        'alias' => 'magiciens'
        ),
        2 => array(
        'alias' => 'hobbits'
        ),
        3 => array(
        'alias' => 'visiteurs'
      ),
  );
    //Faisons une itération et créons les groupes d'ARO
    foreach($groups as $data) {
        //Pensez à faire un appel à create() au moment d'enregistrer dans
        //la boucle...
        $aro->create();
        //Enregistrement des données
        $aro->save($data);
    }
    //Les autres actions logiques seront à placer ici...
  }
```

Une fois que nous avons cela, nous pouvons utiliser la console d'application ACL pour vérifier la structure de l'arbre.

```
$ cake acl view aro

Arbre d' Aro:
---------------------------------------------------------------
  [1]guerriers

  [2]magiciens

  [3]hobbits

  [4]visiteurs

---------------------------------------------------------------
```

Je suppose qu'il n'y en a pas beaucoup dans l'arbre à ce niveau, mais au minimum quelques vérifications que nous avons faites aux quatres noeuds de niveaux supérieurs. Ajoutons quelques enfants à ces noeuds ARO en ajoutant nos AROs utilisateurs dans ces groupes. Tous les bons citoyens de la Terre du Milieu ont un accompte dans notre nouveau système, nous allons alors lier les enregistrements d'ARO aux enregistrements spécifiques du model de notre base de données.

Note : Quand nous ajouterons un noeud enfant à un arbre, nous devrons nous assurer d'utiliser les ID des noeuds ACL, plutôt que d'utiliser la valeur de la foreign_key (clé étrangère).

```php
public function anyAction(){
    $aro = new Aro();
    //Ici nous avons les enregistrement de nos utilisateurs prêts à être liés aux
    //nouveaux enregistrements d'ARO. Ces données peuvent venir d'un model et
    //modifiées, mais nous utiliserons des tableaux statiques pour les besoins de la
    //démonstration.
    $users = array(
        0 => array(
          'alias' => 'Aragorn',
          'parent_id' => 1,
          'model' => 'User',
          'foreign_key' => 2356,
        ),
        1 => array(
          'alias' => 'Legolas',
          'parent_id' => 1,
          'model' => 'User',
          'foreign_key' => 6342,
        ),
        2 => array(
          'alias' => 'Gimli',
          'parent_id' => 1,
          'model' => 'User',
          'foreign_key' => 1564,
        ),
        3 => array(
          'alias' => 'Gandalf',
          'parent_id' => 2,
```

```
            'model' => 'User',
            'foreign_key' => 7419,
        ),
        4 => array(
            'alias' => 'Frodo',
            'parent_id' => 3,
            'model' => 'User',
            'foreign_key' => 7451,
        ),
        5 => array(
            'alias' => 'Bilbo',
            'parent_id' => 3,
            'model' => 'User',
            'foreign_key' => 5126,
        ),
        6 => array(
            'alias' => 'Merry',
            'parent_id' => 3,
            'model' => 'User',
            'foreign_key' => 5144,
        ),
        7 => array(
            'alias' => 'Pippin',
            'parent_id' => 3,
            'model' => 'User',
            'foreign_key' => 1211,
        ),
        8 => array(
            'alias' => 'Gollum',
            'parent_id' => 4,
            'model' => 'User',
            'foreign_key' => 1337,
        ),
    );
    //Faisons une itération et créons les AROs (comme des enfants)
    foreach($users as $data){
        //Pensez à faire un appel à create() au moment d'enregistrer dans
        //la boucle...
        $aro->create();
        //Enregistrement des données
        $aro->save($data);
    }
    //Les autres actions logiques se trouveront ici ...
}
```

Note : Typiquement vous n'aurez pas à fournir et un alias, et un model/clé_étrangère, mais nous les utiliserons ici pour faire une structure d'arbre plus facile à lire pour les besoins de la démonstrations.

La sortie console de cette commande peut maintenant nous intéresser un peu plus. Nous allons faire un essai :

```
  $ cake acl view aro


Arbre d'Aro:
---------------------------------------------------------------
[1]guerriers

  [5]Aragorn

  [6]Legolas

  [7]Gimli

[2]magiciens

  [8]Gandalf

[3]hobbits

  [9]Frodo

  [10]Bilbo

  [11]Merry

  [12]Pippin

[4]visiteurs

  [13]Gollum


  ---------------------------------------------------------------
```

Maintenant que nous avons notre arbre d'ARO configuré proprement, revenons sur une possible approche de structure d'arbre d'ACO. Alors que nous pouvons structurer plus d'une représentation abstraite de nos ACOs, il est parfois plus pratique de modéliser un arbre ACO après le paramètrage du Controller/Action de Cake. Nous avons cinq principaux objets à manipuler dans le scénario de la Communauté, et la configuration naturelle pour cela dans une application Cake est un groupe de models, et tout à la fin les controllers qui les manipulent. A côté des controllers eux-mêmes, nous allons vouloir contrôler l'accès à des actions spécifiques dans ces controllers.

En partant de cette idée, nous allons configurer un arbre d'ACO qui va imiter une configuration d'application Cake. Depuis nos cinq ACOs, nous allons créer un arbre d'ACO qui devra ressembler à ça :
- Armes
- Anneaux
- MorceauxPorc
- EffortsDiplomatiques
- Bières

Une bonne chose concernant la configuration des ACL et que chaque ACO va automatiquement contenir quatre propriétés relatives aux actions CRUD (créer, lire, mettre à jour et supprimer). Vous pouvez créer

des noeuds enfants sous chacun de ces cinq principaux ACOs, mais l'utilisation des actions de management intégrées à Cake permet d'aborder les opérations basiques de CRUD sur un objet donné. Gardez à l'esprit qu'il faudra faire vos arbres d'ACO plus petits et plus faciles à maintenir. Nous allons voir comment ils sont utilisés plus tard quand nous parlerons de la façon d'assigner les permissions.

Nous sommes maintenant des pros de l'ajout d'AROs et de l'utilisation des techniques de création d'arbres d'ACO. La création de groupes d'un niveau supérieur utilise le model Aco du coeur.

Assigner les Permissions Après la création de nos ACOs et AROs, nous pouvons finalement assigner des permissions entre les deux groupes. Ceci est réalisé en utilisant le component Acl du cœur de CakePHP. Continuons avec notre exemple.

Ici nous travaillerons dans un contexte d'une action de controller. Nous faisons cela parce que les permissions sont managées par le component Acl.

```
class SomethingsController extends AppController {
    // Vous pourriez placer çà dans AppController
    // mais cela fonctionne bien ici aussi.

    public $components = array('Acl');

}
```

Configurons quelques permissions de base, en utilisant le Component Acl dans une action à l'intérieur de ce controller.

```
public function index() {
    //Autorise un accès complet aux armes pour les guerriers
    //Ces exemples utilisent tous deux la syntaxe avec un alias
    $this->Acl->allow('guerriers', 'Armes');

    //Encore que le Roi pourrait ne pas vouloir laisser n'importe qui
    //disposer d'un accès sans limites
    $this->Acl->deny('guerrier/Legolas', 'Armes', 'delete');
    $this->Acl->deny('guerrier/Gimli',   'Armes', 'delete');

    die(print_r('done', 1));
}
```

Le premier appel que nous faisons au component Acl donne, à tout utilisateur appartenant au groupe ARO 'guerriers', un accès total à tout ce qui appartient au groupe ACO 'Armes'. Ici nous adressons simplement les ACOs et AROs d'après leurs alias.

Avez-vous noté l'usage du troisième paramètre ? C'est là où nous utilisons ces actions bien pratiques qui sont intégrées à tous les ACOs de Cake. Les options par défaut pour ce paramètre sont `create`, `read`, `update` et `delete`, mais vous pouvez ajouter une colonne dans la table `aros_acos` de la base de données (préfixée avec _ - par exemple `_admin`) et l'utiliser en parallèle de celles par défaut.

Le second ensemble d'appels est une tentative d'affiner les permissions. Nous voulons qu'Aragorn conserve ses privilèges de plein accès, mais nous refusons aux autres guerriers du groupe, la capacité de supprimer

les enregistrements de la table Armes. Nous utilisons la syntaxe avec un alias pour adresser les AROs ci-dessus, mais vous pourriez utiliser votre propre syntaxe model/clé étrangère. Ce que nous avons ci-dessus est équivalent à ceci :

```
// 6342 = Legolas
// 1564 = Gimli

$this->Acl->deny(array('model' => 'Utilisateur', 'foreign_key' => 6342), 'Armes', 'delete')
$this->Acl->deny(array('model' => 'Utilisateur', 'foreign_key' => 1564), 'Armes', 'delete')
```

Note : L'adressage d'un nœud en utilisant la syntaxe avec un alias, nécessite une chaîne délimitée par des slashs ('/utilisateurs/salaries/developpeurs'). L'adressage d'un nœud en utilisant la syntaxe model/clé étrangère nécessite un tableau avec deux paramètres : `array('model' => 'Utilisateur', 'foreign_key' => 8282)`.

La prochaine section nous aidera à valider notre configuration, en utilisant le component Acl pour contrôler les permissions que nous venons de définir.

Vérification des Permissions : le Component ACL Utilisons le Component Acl pour nous assurer que les nains et les elfes ne peuvent déplacer des choses depuis l'armurerie. Maintenant, nous devrions être en mesure d'utiliser le Component Acl, pour faire une vérification entre les ACOs et les AROs que nous avons créés. La syntaxe de base pour faire une vérification des permissions est :

```
$this->Acl->check($aro, $aco, $action = '*');
```

Faisons un essai dans une action de controller :

```
public function index() {
    // Tout cela renvoi true:
    $this->Acl->check('guerriers/Aragorn', 'Armes');
    $this->Acl->check('guerriers/Aragorn', 'Armes', 'create');
    $this->Acl->check('guerriers/Aragorn', 'Armes', 'read');
    $this->Acl->check('guerriers/Aragorn', 'Armes', 'update');
    $this->Acl->check('guerriers/Aragorn', 'Armes', 'delete');


    // Souvenez-vous, nous pouvons utiliser la syntaxe model/clé étrangère
    // pour nos AROs utilisateur
    $this->Acl->check(array('User' => array('id' => 2356)), 'Weapons');

    // Tout cela renvoi true également:
    $result = $this->Acl->check('guerriers/Legolas', 'Armes', 'create');
    $result = $this->Acl->check('guerriers/Gimli', 'Armes', 'read');


    // Mais ceci retourne "false" :
    $result = $this->Acl->check('guerriers/Legolas', 'Armes', 'delete');
    $result = $this->Acl->check('guerriers/Gimli', 'Armes', 'delete');

}
```

L'usage fait ici est démonstratif, mais vous pouvez sans doute voir comment une telle vérification peut être utilisée, pour décider à quel moment autoriser, ou pas, quelque chose à se produire, pour afficher un message d'erreur ou rediriger l'utilisateur vers un login.

Authentification

class `AuthComponent` (*ComponentCollection $collection*, *array $settings = array()*)

Identifier, authentifier et autoriser des utilisateurs constitue une partie courante de nombreuses applications Web. Le component Auth de CakePHP fournit un moyen modulaire d'accomplir cette tâche. Le component Auth vous permet de combiner l'authentification des objets, l'autorisation des objets pour créer un moyen souple pour permettre l'identification et le contrôle des autorisations de l'utilisateur.

Authentification

L'authentification est le processus d'identification des utilisateurs par des identifiants de connexion définis et permet de s'assurer que l'utilisateur est bien celui qu'il prétend être. En général, cela se fait à travers un nom d'utilisateur et un mot de passe, qui sont comparés à une liste d'utilisateurs connus. Dans CakePHP, il y a plusieurs façons intégrées pour l'authentification des utilisateurs enregistrés dans votre application.

- `FormAuthenticate` vous permet d'authentifier les utilisateurs sur la base de formulaire de donnée POST. Habituellement il s'agit d'un formulaire de connexion ou les utilisateurs entrent des informations.
- `BasicAuthenticate` vous permet d'identifier les utilisateurs en utilisant l'authentification Basic HTTP.
- `DigestAuthenticate` vous permet d'identifier les utilisateurs en utilisant l'authentification Digest HTTP.

Par défaut Le component Auth (`AutComponent`) utilise `FormAuthenticate`.

Choisir un type d'Authentification En général, vous aurez envie d'offrir l'authentification par formulaire. C'est le plus facile pour les utilisateurs utilisant un navigateur Web. Si vous construisez une API ou un service web, vous aurez peut-être à envisager l'utilisation de l'authentification de base ou l'authentification Digest. L'élément clé qui différencie l'authentification digest de l'authentification basic est la plupart du temps liée à la façon dont les mots de passe sont gérés. Avec l'authentification basic, le nom d'utilisateur et le mot de passe sont transmis en clair sur le serveur. Cela rend l'authentification de base non appropriée pour des applications sans SSL, puisque vous exposeriez sensiblement vos mots de passe. L'authentification Digest utilise un hachage condensé du nom d'utilisateur, mot de passe, et quelques autres détails. Cela rend l'authentification Digest plus approprié pour des applications sans cryptage SSL.

Vous pouvez également utiliser des systèmes d'authentification comme OpenID, mais openid ne fait pas parti du cœur de CakePHP.

Configuration des gestionnaires d'authentification Vous configurez les gestionnaires d'authentification en utilisant `$this->Auth->authenticate`. Vous pouvez configurer un ou plusieurs gestionnaires pour l'authentification. L'utilisation de plusieurs gestionnaires d'authentification vous permet de supporter les différentes méthodes de connexion des utilisateurs. Quand les utilisateurs se connectent, les gestionnaires d'authentification sont utilisés dans l'ordre auquel ils ont été déclarés. Une fois qu'un gestionnaire est capable d'identifier un utilisateur, les autres gestionnaires ne seront pas utilisés. Inversement, vous pouvez

mettre un terme à tous les authentifications en levant une exception.Vous pourrez traiter toutes les exceptions levées, et les gérer comme désiré.

Vous pouvez configurer le gestionnaire d'authentification dans les tableaux `beforeFilter` ou `$components`. Vous pouvez passer l'information de configuration dans chaque objet d'authentification en utilisant un tableau :

```
// Configuration de base
$this->Auth->authenticate = array('Form');

// Passer la configuration
$this->Auth->authenticate = array(
    'Basic' => array('userModel' => 'Membre'),
    'Form' => array('userModel' => 'Membre')
);
```

Dans le deuxième exemple vous pourrez noter que nous avons à déclarer la clé `userModel` deux fois. Pour vous aider à garder un code "propre", vous pouvez utiliser la clé `all`. Cette clé spéciale vous permet de définir les réglages qui sont passés à chaque objet attaché. La cle `all` est aussi utilisée comme cela `AuthComponent::ALL` :

```
// Passer la configuration en utilisant 'all'
$this->Auth->authenticate = array(
    AuthComponent::ALL => array('userModel' => 'Membre'),
    'Basic',
    'Form'
);
```

Dans l'exemple ci-dessus, à la fois `'Form` et `Basic` prendront les paramétrages définis dans la clé "all". Tous les paramètres transmis à un objet d'authentification particulier remplaceront la clé correspondante dans la clé 'all'. Les objets d'authentification supportent les clés de configuration suivante.

- `fields` Les champs à utiliser pour identifier un utilisateur.
- `userModel` Le nom du model de l'utilisateur, par défaut User.
- `scope` Des conditions supplémentaires à utiliser lors de la recherche et l'authentification des utilisateurs, ex `array('User.is_active' => 1)`.
- `contain` options de Containable lorque l'enregistrement de l'utilisateur est chargé.

Introduit dans la version 2.2.

- `passwordHasher` Classe de hash de mot de passe. Par défaut à `Simple`. Introduit dans la version 2.4.

Configurer différents champs pour l'utilisateur dans le tableau `$components` :

```
// Passer la configuration dans le tableau $components
public $components = array(
    'Auth' => array(
        'authenticate' => array(
            'Form' => array(
                'fields' => array('username' => 'email')
            )
        )
    )
);
```

Ne mettez pas d'autre clés de configuration de Auth(comme authError, loginAction etc). Ils doivent se trouver au même niveau que la clé d'authentification. La configuration ci-dessus avec d'autres configurations ressemblerait à quelque chose comme. :

```
// Passage de paramètre dans le tableau $components
public $components = array(
    'Auth' => array(
        'loginAction' => array(
            'controller' => 'users',
            'action' => 'login',
            'plugin' => 'users'
        ),
        'authError' => 'Pensiez-vous réellement que vous étiez autorisés à voir cela ?',
        'authenticate' => array(
            'Form' => array(
                'fields' => array('username' => 'email')
            )
        )
    )
);
```

En plus de la configuration courante, l'authentification de base prend en charge les clés suivantes :
- `realm` Le domaine en cours d'authentification. Par défaut à `env('SERVER_NAME')`.

En plus de la configuration courante, l'authentification Digest prend en charge les clés suivantes :
- `realm` Le domaine en cours d'authentification. Par défaut à servername
- `nonce` Un nonce utiliser pour l'authentification. Par défaut à `uniqid()`.
- `qop` Par défaut à auth, pas d'autre valeur supportée pour le moment.
- `opaque` Une chaîne qui doit être retourné à l'identique par les clients. Par Défaut à `md5($settings['realm'])`.

Identifier les utilisateurs et les connecter Par le passé le component Auth `AuthComponent` connectait les utilisateurs automatiquement. C'était un peu déroutant pour certain, et rendait la création au travers du component Auth `AuthComponent` par moment un peu difficile. Avec la version 2.0, vous avez besoin d'appeler manuellement `$this->Auth->login()` pour connecter un utilisateur.

Quand les utilisateurs s'identifient, les objets d'identification sont vérifiés dans l'ordre où ils ont été attachés. Une fois qu'un objet peut identifier un utilisateur, les autres objets ne sont pas vérifiés. Une simple fonction de connexion pourrait ressembler à cela

```
public function login() {
    if ($this->request->is('post')) {
        if ($this->Auth->login()) {
            return $this->redirect($this->Auth->redirectUrl());
            // Avant 2.3, utilisez 'return $this->redirect($this->Auth->redirect());'
        } else {
            $this->Session->setFlash(__('Username ou password est incorrect'), 'default', a
        }
    }
}
```

Le code ci-dessus (sans aucune donnée transmise à la méthode `login`), tentera de connecter un utilisateur en utilisant les données POST, et sera redirigé en cas de succès sur la dernière page visitée, ou `AuthComponent::$loginRedirect`. Si le login est en échec, un message flash est défini.

Warning : Dans la version 2.0 `$this->Auth->login($this->request->data)` connectera l'utilisateur avec les données postées., tandis que avec la version 1.3 `$this->Auth->login($this->data)` tentera d'identifier l'utilisateur en premier et le connectera seulement en cas de succès.

Utilisation de l'authentification Digest et Basic pour la connexion Puisque les authentifications basic et digest ne nécessitent pas un POST initial ou un form, ainsi si vous utilisez seulement les authentificators basic / digest, vous n'avez pas besoin d'action login dans votre controller. Aussi, vous pouvez définir `AuthComponent::$sessionKey` à false pour vous assurer que AuthComponent n'essaie pas de lire les infos de l'user à partir des sessions. L'authentification stateless va re-vérifier les certificats de l'user à chaque requête, cela crée un petit montant de charges supplémentaires, mais permet aux clients de se connecter sans utiliser les cookies.

Note : Avant 2.4, vous avez toujours besoin de l'action login puisque vous êtes redirigés vers login quand un user non authentifié essaie d'accéder à une page protégée même en utilisant seulement l'auth basic ou digest. Aussi configurer `AuthComponent::$sessionKey` à false va causer une erreur avant 2.4.

Créer des objets d'authentification personnalisés Comme les objets d'authentification sont modulaires, vous pouvez créer des objets d'authentification personnalisés pour votre application ou plugins. Si par exemple vous vouliez créer un objet d'authentification OpenID. Dans `app/Controller/Component/Auth/OpenidAuthenticate.php` vous pourriez mettre ce qui suit :

```
App::uses('BaseAuthenticate', 'Controller/Component/Auth');

class OpenidAuthenticate extends BaseAuthenticate {
    public function authenticate(CakeRequest $request, CakeResponse $response) {
        // Faire les trucs d'OpenID ici.
        // Retourne un tableau de l\'user si ils peuvent authentifier
        // l\'user
        // retourne false dans le cas contraire
    }
}
```

Les objets d'authentification devraient retourner `false` si ils ne peuvent identifier l'utilisateur. Et un tableau d'information utilisateur si ils le peuvent.Il n'est pas utile d'étendre (extend) `BaseAuthenticate`, simplement votre objet d'identification doit implémenter la méthode `authenticate()`. La class `BaseAuthenticate` fournie un nombre de méthode très utiles communément utilisées. Vous pouvez aussi implémenter une méthode `getUser()` si votre objet d'identification doit supporter des authentifications sans cookie ou sans état (stateless). Regardez les sections portant sur l'authentification digest et basic plus bas pour plus d'information.

Utilisation d'objets d'authentification personnalisés Une fois votre objet d'authentification créer, vous pouvez les utiliser en les incluant dans le tableau d'authentification AuthComponents :

```
$this->Auth->authenticate = array(
    'Openid', // objet d'authentification app
    'AuthBag.Combo', // plugin objet d'identification.
);
```

Création de systèmes d'authentification stateless Les objets d'authentification peuvent implémenter une méthode `getUser()` qui peut être utilisée pour supporter les systèmes de connexion des utilisateurs qui ne reposent pas sur les cookies. Une méthode getUser typique regarde l'environnement de la requête (request/environnement) et y utilise les informations d'identification de l'utilisateur. L'authentification HTTP Basic utilise par exemple `$_SERVER['PHP_AUTH_USER']` et `$_SERVER['PHP_AUTH_PW']` pour les champs username et password. Pour chaque requête, si un client ne supporte pas les cookies, ces valeurs sont utilisées pour ré-identifier l'utilisateur et s'assurer que c'est un utilisateur valide. Comme avec les méthodes d'authentification de l'objet `authenticate()`, la méthode `getuser()` devrait retourner un tableau d'information utilisateur en cas de succès, et `false` en cas d'echec. :

```
public function getUser($request) {
    $username = env('PHP_AUTH_USER');
    $pass = env('PHP_AUTH_PW');

    if (empty($username) || empty($pass)) {
        return false;
    }
    return $this->_findUser($username, $pass);
}
```

Le contenu ci-dessus montre comment vous pourriez mettre en œuvre la méthode getUser pour les authentifications HTTP Basic. La méthode `_findUser()` fait partie de `BaseAuthenticate` et identifie un utilisateur en se basant sur un nom d'utilisateur et un mot de passe.

Gestion des requêtes non authentifiées Quand un user non authentifié essaie d'accéder à une page protégée en premier, la méthode *unauthenticated()* du dernier authenticator dans la chaîne est appelée. L'objet d'authentification peut gérer la réponse d'envoi ou la redirection appropriée et retourne *true* pour indiquer qu'aucune action suivante n'est nécessaire. Du fait de l'ordre dans lequel vous spécifiez l'objet d'authentification dans les propriétés de *AuthComponent : :$authenticate*.

Si authenticator retourne null, *AuthComponent* redirige l'user vers l'action login. Si c'est une requête ajax et *AuthComponent : :$ajaxLogin* est spécifiée, cet element est rendu, sinon un code de statut http 403 est retourné.

Note : Avant 2.4, les objets d'authentification ne fournissent pas de méthode *unauthenticated()*.

Afficher les messages flash de Auth Pour afficher les messages d'erreur de session que Auth génère, vous devez ajouter les lignes de code suivante dans votre layout. Ajoutez les deux lignes suivantes au

fichier `app/View/Layouts/default.ctp` dans la section body de préférence avant la ligne content_for_layout. :

```
echo $this->Session->flash();
echo $this->Session->flash('auth');
```

Vous pouvez personnaliser les messages d'erreur, et les réglages que le component Auth `AuthComponent` utilise. En utilisant `$this->Auth->flash` vous pouvez configurer les paramètres que le component Auth utilise pour envoyer des messages flash. Les clés disponibles sont :
– `element` - L'élément à utiliser , 'default' par défaut.
– `key` - La clé a utiliser , 'auth' par défaut
– `params` - Le tableau des paramètres additionnels à utiliser, array() par défaut
En plus des paramètres de message flash, vous pouvez personnaliser d'autres messages d'erreurs que le component AuthComponent utilise. Dans la partie beforeFilter de votre controller, ou dans le paramétrage du component vous pouvez utiliser `authError` pour personnaliser l'erreur à utiliser quand l'authentification échoue

```
$this->Auth->authError = "Cette erreur se présente à l'utilisateur qui tente d'accéder à un
```

Modifié dans la version 2.4 : Parfois, vous voulez seulement afficher l'erreur d'autorisation après que l'user se soit déja connecté. Vous pouvez supprimer ce message en configurant sa valeur avec le boléen *false*. Dans le beforeFilter() de votre controller, ou les configurations du component :

```
if (!$this->Auth->loggedIn()) {
    $this->Auth->authError = false;
}
```

Hachage des mots de passe Le component Auth ne fait fait plus automatiquement le hachage de tous les mots de passe qu'il rencontre. Ceci à été enlevé parce qu'il rendait un certain nombre de tâches communes comme la validation difficile. Vous ne devriez **jamais** stocker un mot de passe en clair, et avant de sauvegarder un utilisateur vous devez toujours hacher le mot de passe.

As of 2.4 the generation and checking of password hashes has been delegated to password hasher classes. Authenticating objects use a new setting `passwordHasher` which specifies the password hasher class to use. It can be a string specifying class name or an array with key `className` stating the class name and any extra keys will be passed to password hasher constructor as config. The default hasher class `Simple` can be used for sha1, sha256, md5 hashing. By default the hash type set in Security class will be used. You can use specific hash type like this :

```
public $components = array(
    'Auth' => array(
        'authenticate' => array(
            'Form' => array(
                'passwordHasher' => array(
                    'className' => 'Simple',
                    'hashType' => 'sha256'
                )
            )
        )
```

```
    )
);
```

When creating new user records you can hash a password in the beforeSave callback of your model using appropriate password hasher class :

```
App::uses('SimplePasswordHasher', 'Controller/Component/Auth');

class User extends AppModel {
    public function beforeSave($options = array()) {
        if (!$this->id) {
            $passwordHasher = new SimplePasswordHasher();
            $this->data['User']['password'] = $passwordHasher->hash($this->data['User']['pa
        }
        return true;
    }
}
```

Vous n'avez pas besoin de hacher le mot de passe avant d'appeler `$this->Auth->login()`. Les différents objets d'authentification hacherons les mots de passe individuellement.

Utiliser bcrypt pour les mots de passe Dans CakePHP 2.3, la classe `BlowfishAuthenticate` a été introduite pour permettre l'utilisation de bcrypt (https ://en.wikipedia.org/wiki/Bcrypt) c'est-à-dire Blowfish pour les mots de passe hashés. Les hashes Bcrypt sont plus difficiles à forcer sauvagement par rapport aux mots de passe stockés avec sha1. Mais `BlowfishAuthenticate` a été déprécié dans 2.4 et à la place `BlowfishPasswordHasher` a été ajoutée.

Un hasher de mot de passe blowfish peut être utilisé avec toute classe d'authentification. Tout ce que vous avez à faire est de spécifier la configuration `passwordHasher` pour l'objet d'authentification :

```
public $components = array(
    'Auth' => array(
        'authenticate' => array(
            'Form' => array(
                'passwordHasher' => 'Blowfish'
            )
        )
    )
);
```

Hachage de mots de passe pour l'authentification Digest Puisque l'authentification Digest nécessite un mot de passe haché dans un format défini par la RFC. Respectivement pour hacher correctement un mot de passe pour l'utilisation de l'authentification Digest vous devriez utilisez la fonction spéciale `DigestAuthenticate`. Si vous vous apprêtez à combiner l'authentification Digest avec d'autres stratégies d'authentifications, il est aussi recommandé de stocker le mot de passe Digest dans une colonne séparée, pour le hachage normal de mot de passe :

```
class User extends AppModel {
    public function beforeSave($options = array()) {
        // fabrique un mot de passe pour l'auth Digest.
```

```
        $this->data['User']['digest_hash'] = DigestAuthenticate::password(
            $this->data['User']['username'], $this->data['User']['password'], env('SERVER_N
        );
        return true;
    }
}
```

Les mots de passe pour l'authentification Digest ont besoin d'un peu plus d'information que pour d'autres mots de passe hachés. Si vous utilisez le component AuthComponent : :password() pour le hachage Digest vous ne pourrez pas vous connecter.

Note : le troisième paramètre de DigestAuthenticate : :password() doit correspondre à la valeur de la configuration 'realm' définie quand DigestAuthentication était configuré dans AuthComponent : :$authenticate. Par défaut à `env('SCRIPT_NAME)`. Vous devez utiliser une chaîne statique si vous voulez un hachage permanent dans des environnements multiples.

Creating custom password hasher classes Custom password hasher classes need to extend the `AbstractPasswordHasher` class and need to implement the abstract methods `hash()` and `check()`. In `app/Controller/Component/Auth/CustomPasswordHasher.php` you could put the following :

```
App::uses('CustomPasswordHasher', 'Controller/Component/Auth');

class CustomPasswordHasher extends AbstractPasswordHasher {
    public function hash($password) {
        // stuff here
    }

    public function check($password, $hashedPassword) {
        // stuff here
    }
}
```

Connecter les utilisateurs manuellement Parfois, le besoin se fait sentir de connecter un utilisateur manuellement, par exemple juste après qu'il se soit enregistré dans votre application. Vous pouvez faire cela en appelant `$this->Auth->login()` avec les données utilisateur que vous voulez pour la 'connexion' :

```
public function register() {
    if ($this->User->save($this->request->data)) {
        $id = $this->User->id;
        $this->request->data['User'] = array_merge($this->request->data['User'], array('id'
        $this->Auth->login($this->request->data['User']);
        $this->redirect('/users/home');
    }
}
```

Warning : Soyez certain d'ajouter manuellement le nouveau User id au tableau passé à la méthode de login. Sinon vous n'aurez pas l'id utilisateur disponible.

Accéder à l'utilisateur connecté Une fois que l'utilisateur est connecté, vous avez souvent besoin d'information particulière à propos de l'utilisateur courant. Vous pouvez accéder à l'utilisateur en cours de connexion en utilisant `AuthComponent::user()`. Cette méthode est statique, et peut être utilisée globalement après le chargement du component Auth. Vous pouvez y accéder à la fois avec l'instance d'une méthode ou comme une méthode statique :

```
// Utilisez n'importe où
AuthComponent::user('id')

// Depuis l'intérieur du controler
$this->Auth->user('id');
```

Déconnexion des utilisateurs Éventuellement vous aurez besoin d'un moyen rapide pour dés-authentifier les utilisateurs et les rediriger ou il devraient aller. Cette méthode est aussi très pratique si vous voulez fournir un lien 'Déconnecte moi' à l'intérieur de la zone membres de votre application

```
public function logout() {
    $this->redirect($this->Auth->logout());
}
```

La déconnexion des utilisateurs connectés avec l'authentification Basic ou Digest est difficile à accomplir pour tous les clients. La plupart des navigateurs retiennent les autorisations pendant qu'il restent ouvert. Certains navigateurs peuvent être forcés en envoyant un code 401. Le changement du realm de l'authentification est une autre solution qui fonctionne pour certain clients.

Autorisation

l'autorisation est le processus qui permet de s'assurer qu'un utilisateur identifier/authentifier est autorisé à accéder aux ressources qu'il demande. Il y a plusieurs gestionnaires d'autorisation intégrés, et vous pouvez créer vos propres gestionnaires dans un plugin par exemple.

- `ActionsAuthorize` Utilise le Component AclComponent pour vérifier les permissions d'un niveau d'action.
- `CrudAuthorize` Utilise le Component Acl et les action -> CRUD mappings pour vérifier les permissions pour les ressources.
- `ControllerAuthorize` appelle `isAuthorized()` sur le controller actif, et utilise ce retour pour autoriser un utilisateur. C'est souvent le moyen le plus simple d'autoriser les utilisateurs.

Configurer les gestionnaires d'autorisation Vous configurez les gestionnaires d'autorisation en utilisant `$this->Auth->authorize`. Vous pouvez configurer un ou plusieurs gestionnaires . L'utilisation de plusieurs gestionnaires vous donnes la possibilité d'utiliser plusieurs moyens de vérifier les autorisations. Quand les gestionnaires d'autorisation sont vérifiés ils sont appelés dans l'ordre ou ils sont déclarés. Les gestionnaires devraient retourner false, s'il ne sont pas capable de vérifier les autorisation, ou bien si la

vérification a échouée. Le gestionnaire devrait retourner true si ils sont capables de vérifier correctement les autorisations. Les gestionnaires seront appelés dans l'ordre jusqu'à ce qu'un passe. Si toutes les vérifications échoues , l'utilisateur sera redirigé vers la page d'où il vient. Vous pouvez également stopper les autorisations en levant une exception. Vous aurez besoin de traiter toutes les exceptions levées, et les manipuler.

Vous pouvez configurer les gestionnaires d'autorisation dans le `beforeFilter` de votre controller ou , dans le tableau `$components`. Vous pouvez passer les informations de configuration dans chaque objet d'autorisation, en utilisant un tableau :

```
// paramétrage Basique
$this->Auth->authorize = array('Controller');

// passage de paramètre
$this->Auth->authorize = array(
    'Actions' => array('actionPath' => 'controllers/'),
    'Controller'
);
```

Tout comme `Auth->authenticate`, `Auth->authorize`, vous aident à garder un code "propre, en utilisant la clé `all`. Cette clé spéciale vous aide à définir les paramètres qui sont passés à chaque objet attaché. La clé all est aussi exposée comme `AuthComponent::ALL` :

```
// passage de paramètre en utilisant 'all'
$this->Auth->authorize = array(
    AuthComponent::ALL => array('actionPath' => 'controllers/'),
    'Actions',
    'Controller'
);
```

Dans l'exemple ci-dessus, à la fois `L' Actions` et `Le Controller` auront les paramètres définis pour la clé 'all'. Chaque paramètres passés a un objet d'autorisation spécifique remplacera la clé correspondante dans la clé 'all'. Le noyau authorize objects supporte les clés de configuration suivantes.

- `actionPath` Utilisé par `ActionsAuthorize` pour localiser le controller action ACO's dans l'arborescence ACO.
- `actionMap` Action -> CRUD mappings. Utilisé par `CrudAuthorize` et les objets d'autorisation qui veulent mapper les actions aux rôles CRUD.
- `userModel` Le nom du nœud ARO/Model dans lequel l'information utilisateur peut être trouvé. Utilisé avec ActionsAuthorize.

Création d'objets Authorize personnalisés Parce que les objets authorize sont modulables, vous pouvez créer des objets authorize personnalisés dans votre application, ou plugins. Si par exemple vous voulez créer un objet authorize LDAP. Dans `app/Controller/Component/Auth/LdapAuthorize.php`, vous pourriez mettre cela :

```
App::uses('BaseAuthorize', 'Controller/Component/Auth');

class LdapAuthorize extends BaseAuthorize {
    public function authorize($user, CakeRequest $request) {
        // Faire les trucs pour le LDAP ici.
    }
}
```

L'objet Authorize devrait retourner *false ' si l'utilisateur se voit refuser l'accès, ou si l'objet est incapable de faire un contrôle. Si l'objet est capable de vérifier les accès de l'utilisateur, ''true'* devrait être retourné. Ça n'est pas nécessaire d'étendre `BaseAuthorize`, il faut simplement que votre objet authorize implémente la méthode `authorize()`. La classe `BaseAuthorize` fournit un nombre intéressant de méthodes utiles qui sont communément utilisées.

Utilisation d'objets Authorize personnalisés Une fois que vous avez créé votre objet authorize personnalisé, vous pouvez l'utiliser en l'incluant dans le tableau authorize :

```
$this->Auth->authorize = array(
    'Ldap', // objet app authorize .
    'AuthBag.Combo', // objet authorize du plugin.
);
```

Ne pas utiliser d'autorisation Si vous souhaitez ne pas utiliser les objets d'autorisation intégrés, et que vous voulez gérer les choses entièrement à l'extérieur du Component Auth (AuthComponent) vous pouvez définir `$this->Auth->authorize = false;`. Par défaut le component Auth démarre avec `authorize = false`. Si vous n'utilisez pas de schéma d'autorisation, assurez-vous de vérifier les autorisations vous-même dans la partie beforeFilter de votre controller ou avec un autre component.

Rendre des actions publiques Il y a souvent des actions de controller que vous souhaitez laisser entièrement publiques, ou qui ne nécessitent pas de connexion utilisateur. Le component Auth (AuthComponnent) est pessimiste, et par défaut interdit l'accès. Vous pouvez marquer des actions comme publique en utilisant `AuthComponent::allow()`. En marquant les actions comme publique, le component Auth ne vérifiera pas la connexion d'un utilisateur, ni n'autorisera la vérification des objets

```
// Permet toutes les actions. CakePHP 2.0
$this->Auth->allow('*');

// Permet toutes les actions. CakePHP 2.1
$this->Auth->allow();

// Ne permet que les actions view et index.
$this->Auth->allow('view', 'index');

// Ne permet que les actions view et index.
$this->Auth->allow(array('view', 'index'));
```

> **Warning :** Si vous utilisez le scaffolding, permettre tout ne vas identifier et autoriser les méthodes scaffoldées. Vous devez spécifier les noms des actions.

Vous pouvez fournir autant de nom d'action dont vous avez besoin à `allow()`. Vous pouvez aussi fournir un tableau contenant tous les noms d'action.

Fabriquer des actions qui requièrent des autorisations Par défaut, toutes les actions nécessitent une authorisation. Cependant, si après avoir rendu les actions publiques, vous voulez révoquer les accès publics. Vous pouvez le faire en utilisant `AuthComponent::deny()` :

```
// retire une action
$this->Auth->deny('add');

// retire toutes les actions .
$this->Auth->deny();

// retire un groupe d'actions.
$this->Auth->deny('add', 'edit');
$this->Auth->deny(array('add', 'edit'));
```

Vous pouvez fournir autant de noms d'action que vous voulez à `deny()`. Vous pouvez aussi fournir un tableau contenant tous les noms d'action.

Utilisation de ControllerAuthorize ControllerAuthorize vous permet de gérer les vérifications d'autorisation dans le callback d'un controller. C'est parfait quand vous avez des autorisations très simples, ou que vous voulez utiliser une combinaison models + components à faire pour vos autorisations, et ne voulez pas créer un objet authorize personnalisé.

Le callback est toujours appelé `isAuthorized()` et devrait retourner un booléen pour indiquer si l'utilisateur est autorisé ou pas à accéder aux ressources de la requête. Le callback est passé à l'utilisateur actif, il peut donc être vérifié :

```
class AppController extends Controller {
    public $components = array(
        'Auth' => array('authorize' => 'Controller'),
    );
    public function isAuthorized($user = null) {
        // Chacun des utilisateur enregistré peut accéder aux fonctions publiques
        if (empty($this->request->params['admin'])) {
            return true;
        }

        // Seulement les administrateurs peuvent accéder aux fonctions d'administration
        if (isset($this->request->params['admin'])) {
            return (bool)($user['role'] === 'admin');
        }

        // Par défaut n'autorise pas
        return false;
    }
}
```

Le callback ci-dessus fournirait un système d'autorisation très simple où seuls les utilisateurs ayant le rôle d'administrateur pourraient accéder aux actions qui ont le préfixe admin.

Utilisation de ActionsAuthorize ActionsAuthorize s'intègre au component ACL, et fournit une vérification ACL très fine pour chaque requête. ActionsAuthorize est souvent jumelé avec DbAcl pour apporter un système de permissions dynamique et flexible qui peuvent être éditées par les utilisateurs administrateurs au travers de l'application. Il peut en outre être combiné avec d'autres implémentations Acl comme IniAcl et des applications Acl backends personnalisées.

Utilisation de CrudAuthorize `CrudAuthorize` s'intègre au component Acl, et fournit la possibilité de mapper les requêtes aux opérations CRUD. Fournit la possibilité d'autoriser l'utilisation du mapping CRUD. Les résultats mappés sont alors vérifiés dans le component Acl comme des permissions spécifiques.

Par exemple, en prenant la requête `/posts/index`. Le mapping par défaut pour *index* est une vérification de la permission de `read`. La vérification d'Acl se ferait alors avec les permissions de `read``pour le controller ``posts`. Ceci vous permet de créer un système de permission qui met d'avantage l'accent sur ce qui est en train d'être fait aux ressources, plutôt que sur l'action spécifique en cours de visite.

Mapper les actions en utilisant CrudAuthorize Quand vous utilisez CrudAuthorize ou d'autres objets authorize qui utilisent le mapping d'action, il peut être nécessaire de mapper des méthodes supplémentaires. vous pouvez mapper des actions –> CRUD permissions en utilisant mapAction(). En l'appelant dans le component Auth vous déléguerez toutes les actions aux objets authorize configurés, ainsi vous pouvez être sûr que le paramétrage sera appliqué partout :

```
$this->Auth->mapActions(array(
    'create' => array('register'),
    'view' => array('show', 'display')
));
```

La clé pour mapActions devra être les permissions CRUD que vous voulez définir, tandis que les valeurs devront être un tableau de toutes les actions qui sont mappées vers les permissions CRUD.

API de AuthComponent

Le component Auth est l'interface primaire à la construction de mécanisme d'autorisation et d'authentification intégrée dans CakePHP.

property `AuthComponent::$`**`ajaxLogin`**
Le nom d'une vue optionnelle d'un élément à rendre quand une requête Ajax est faite avec une session expirée invalide.

property `AuthComponent::$`**`authenticate`**
Défini comme un tableau d'objets d'identifications que vous voulez utiliser quand les utilisateurs de connectent. Il y a plusieurs objets d'authentification dans le noyau, cf la section *Authentification*

property `AuthComponent::$`**`authError`**
Erreur à afficher quand les utilisateurs font une tentative d'accès à un objet ou une action à laquelle ils n'ont pas accès. Modifié dans la version 2.4 : You can suppress authError message from being displayed by setting this value to boolean *false*.

property `AuthComponent::$`**`authorize`**
Défini comme un tableau d'objets d'autorisation que vous voulez utiliser quand les utilisateurs sont autorisés sur chaque requête, cf la section *Autorisation*

property `AuthComponent::$`**`components`**
D'autre components utilisés par le component Auth.

property `AuthComponent::$`**`flash`**
Paramétrage à utiliser quand Auth à besoin de faire un message flash avec `SessionComponent::setFlash()`. Les clés disponibles sont :

–`element` - L'élement à utiliser , par défaut à 'default'.
–`key` - La clé à utiliser, par défaut à 'auth'.
–`params` - Un tableau de paramètres supplémentaires à utiliser par défaut à array()

property `AuthComponent::$loginAction`
Une URL (définie comme une chaîne de caractères ou un tableau) pour l'action du controller qui gère les connexions. Par défaut à */users/login*.

property `AuthComponent::$loginRedirect`
L' URL (définie comme une chaîne de caractères ou un tableau) pour l'action du controller où les utilisateurs doivent être redirigés après la connexion. Cette valeur sera ignorée si l'utilisateur à une valeur `Auth.redirect` dans sa session.

property `AuthComponent::$logoutRedirect`
L'action par défaut pour rediriger l'utilisateur quand il se déconnecte. Alors que le component Auth ne gère pas les redirection post-logout, une URL de redirection sera retournée depuis `AuthComponent::logout()`. Par défaut à `AuthComponent::$loginAction`.

property `AuthComponent::$unauthorizedRedirect`
Contrôle la gestion des accès non autorisés. Par défaut, un utilisateur non autorisé est redirigé vers l'url référente ou vers `AuthComponent::$loginAction` ou '/'. Si défini à false, une exception ForbiddenException est lancée au lieu de la redirection.

property `AuthComponent::$request`
Objet Requête

property `AuthComponent::$response`
Objet Réponse

property `AuthComponent::$sessionKey`
Le nom de la clé de session où les enregistrements de l'utilisateur actuel sont enregistrés. Si ça n'est pas spécifié, ce sera "Auth.User".

`AuthComponent::allow`(*$action*[, *$action*, ...])
Définit une ou plusieurs actions comme publiques, cela signifie qu'aucun contrôle d'autorisation ne sera effectué pour les actions spécifiées. La valeur spéciale `'*'` marquera les actions du controller actuelle comme publique. Sera mieux utilisé dans la méthode beforeFilter de votre controller.

`AuthComponent::constructAuthenticate()`
Charge les objets d'authentification configurés.

`AuthComponent::constructAuthorize()`
Charge les objets d'autorisation configurés.

`AuthComponent::deny`(*$action*[, *$action*, ...])
Basculer une ou plusieurs actions précédemment déclarées comme publique en méthodes non publiques. Ces méthodes requièrent une authorization. Sera mieux utilisé dans la méthode beforeFilter de votre controller.

`AuthComponent::flash`(*$message*)
Définit un message flash. Utilise le component Session, et prend les valeurs depuis `AuthComponent::$flash`.

`AuthComponent::identify`(*$request*, *$response*)

Paramètres
- **$request** (*CakeRequest*) – La requête à utiliser.
- **$response** (*CakeResponse*) – La réponse à utiliser, les en-tête peuvent être envoyées si l'authentification échoue.

Cette méthode est utilisée par le component Auth pour identifier un utilisateur en se basant sur les informations contenues dans la requête courante.

`AuthComponent::initialize`(*$Controller*)
Initialise le component Auth pour une utilisation dans le controller.

`AuthComponent::isAuthorized`(*$user = null*, *$request = null*)
Utilise les adaptateurs d'autorisation configurés pour vérifier qu'un utilisateur est configuré ou non. Chaque adaptateur sera vérifié dans l'ordre, si chacun d'eux retourne true, alors l'utilisateur sera autorisé pour la requête.

`AuthComponent::loggedIn`()
Retourne true si le client actuel est un utilisateur connecté, ou false si il ne l'est pas.

`AuthComponent::login`(*$user*)

Paramètres
- **$user** (*array*) – Un tableau de données d'utilisateurs connectés.

Prends un tableau de données de l'utilisateur pour se connecter. Permet la connexion manuelle des utilisateurs. L'appel de user() va renseigner la valeur de la session avec les informations fournies. Si aucun utilisateur n'est fourni, le component Auth essaiera d'identifier un utilisateur en utilisant les informations de la requête en cours. cf `AuthComponent::identify()`.

`AuthComponent::logout`()

Retourne Une chaîne url où rediriger l'utilisateur déconnecté.

Déconnecte l'utilisateur actuel.

`AuthComponent::mapActions`(*$map = array()*)
Mappe les noms d'action aux opérations CRUD. Utilisé par les authentifications basées sur le controller. Assurez-vous d'avoir configurée la propriété authorize avant d'appeler cette méthode. Ainsi cela déléguera $map à tous les objets autorize attachés.

static `AuthComponent::password`(*$pass*)

Obsolète depuis la version 2.4.

`AuthComponent::redirect`(*$url = null*)

Obsolète depuis la version 2.3.

`AuthComponent::redirectUrl`(*$url = null*)
Si il n'y a pas de paramètre passé, elle obtient l'authentification de redirection de l'URL. Passe une URL pour définir la destination ou un utilisateur devrait être redirigé lors de la connexion. Se repliera vers `AuthComponent::$loginRedirect` si il n'y a pas de valeur de redirection stockée.

Introduit dans la version 2.3.

`AuthComponent::shutdown`(*$Controller*)
Component shutdown. Si un utilisateur est connecté, liquide la redirection.

`AuthComponent::startup`(*$Controller*)

Méthode d'exécution principale. Gère la redirection des utilisateurs invalides et traite les données des formulaires de connexion.

static `AuthComponent::user`(*$key = null*)

Paramètres

– **$key** (*string*) – La clé des données utilisateur que vous voulez récupérer.Si elle est null, tous les utilisateurs seront retournés. Peut aussi être appelée comme une instance de méthode.

Prend les données concernant de l'utilisateur connecté, vous pouvez utiliser une clé propriétaire pour appeler une donnée spécifique à propos d'un utilisateur :

```
$id = $this->Auth->user('id');
```

Si l'utilisateur courant n'est pas connecté ou que la clé n'existe pas `null` sera retourné.

Cookie

class `CookieComponent`(*ComponentCollection $collection*, *array $settings = array()*)

Le component Cookie est un conteneur de la méthode native de PHP `setcookie`. Il inclut également toutes sortes de fonctionnalités pour rendre le codage de code pour les cookies très pratique. Avant de tenter d'utiliser le component Cookie, vous devez vous assurer que 'Cookie'est listé dans la partie $components de votre controller.

Paramétrage du controller

Voici un certain nombre de variables du controller qui vous permettent de configurer la manière dont les cookies sont créés et gérés. Définir ces variables spéciales dans la méthode beforeFilter() de votre controller vous permet de définir la façon dont le Component cookie fonctionne.

variable Cookie	par défaut	description
string $name	'CakeCookie'	Le nom du cookie
string $key	null	Cette chaîne de caractère est utilisée pour chiffrer la valeur écrite dans le cookie. Cette chaîne devrait être aléatoire et difficile à deviner. Quand on utilise le chiffrement rijndael, cette valeur doit être plus grande que 32 bytes.
string $domain	''	Le nom de domaine autoriser à accéder au cookie ex : Utiliser '.votredomaine.com' pour autoriser les accès depuis tout vos sous-domaines
int or string $time	'5 Days'	Le moment ou votre cookie expirera. Les entiers sont Interpretés comme des secondes et une valeur de 0 est équivalente à une 'session cookie' :ex. le cookie expire quand le navigateur est fermé. Si une chaîne est définie ce sera interprété avec la fonction PHP strtotime(). Vous pouvez définir cela a l'intérieur de la méthode write().
string $path	'/'	Le chemin d'accès au server sur lequel le cookie sera appliqué. Si $path est paramétré à '/foo/', il ne sera disponible que dans le repertoires /foo/ et tous les sous repertoires comme /foo/bar/ de votre domaine La valeur par défaut est le domaine entier. Vous pouvez définir cela directement à l'intérieur de la méthode write().
boolean $secure	false	Indique que le cookie ne devrait être transmis qu'au travers une connexion HTTPS sécurisée. Quand cela est défini à true, le cookie ne sera défini que si une connexion sécurisé existe.Vous pouvez définir cela directement à l'intérieur de la méthode write()
boolean $httpOnly	false	Défini à true pour fabriquer uniquement des cookies HTTP. Les cookies seulement HTTP ne sont pas disponibles en javascript

Les extraits de code de controller suivants montrent comment inclure le component Cookie et paramétrer les variables du controller nécessaires pour écrire un cookie nommé 'baker_id' pour le domaine 'example.com' qui a besoin d'une connexion sécurisée, qui est disponible au chemin '/bakers/preferences/', qui expire dans une heure, et est uniquement en HTTP.

```
public $components = array('Cookie');
public function beforeFilter() {
    parent::beforeFilter();
    $this->Cookie->name = 'baker_id';
    $this->Cookie->time = 3600;  // ou '1 hour'
    $this->Cookie->path = '/bakers/preferences/';
    $this->Cookie->domain = 'example.com';
    $this->Cookie->secure = true;  // ex. seulement envoyé si on utilise un HTTPS sécurisé
    $this->Cookie->key = 'qSI232qs*&sXOw!adre@34SAv!@*(XSL#$%)asGb$@11~_+!@#HKis~#^';
    $this->Cookie->httpOnly = true;
}
```

Ensuite, regardons comment utiliser les différentes méthodes du Component Cookie.

Utiliser le Component

Le Component Cookie offre plusieurs méthodes pour travailler avec les Cookies.

`CookieComponent::write`(*mixed $key*, *mixed $value = null*, *boolean $encrypt = true*, *mixed $expires = null*)

La méthode write() est le cœur du composant cookie, $key est le nom de la variable désirée, et $value est l'information à stocker :

```
$this->Cookie->write('nom', 'Rémy');
```

Vous pouvez également grouper vos variables en utilisant la notation point '.' dans les paramètres de clé :

```
$this->Cookie->write('User.name', 'Larry');
$this->Cookie->write('User.role', 'Lead');
```

Si vous voulez écrire plus d'une valeur dans le cookie en une fois, vous pouvez passer un tableau :

```
$this->Cookie->write('User',
    array('name' => 'Larry', 'role' => 'Lead')
);
```

Toutes les valeurs dans le cookie sont chiffrées par défaut. Si vous voulez stocker vos valeurs en texte clair, definissez le troisème paramètre de la méthode write() à false. Le chiffrement utilisé sur les valeurs de cookie est un système de chiffrement très simple. Il utilise `Security.salt` et une variable de classe de configuration prédéfinie `Security.cipherSeed` pour chiffrer les valeurs. Vous devriez changer `Security.cipherSeed` dans app/Config/core.php pour assurer un meilleur chiffrement. :

```
$this->Cookie->write('name', 'Larry', false);
```

Le dernier paramètre à écrire est $expires - le nombre de secondes avant que le cookie n'expire. Par convention, ce paramètre peut aussi être passé comme une chaîne de texte que la fonction strtotime() de php comprend :

```
// Les deux cookies expirent dans une heure.
$this->Cookie->write('first_name', 'Larry', false, 3600);
$this->Cookie->write('last_name', 'Masters', false, '1 hour');
```

`CookieComponent::read`(*mixed $key = null*)

Cette méthode est utilisée pour lire la valeur d'une variable de cookie avec le nom spécifié dans $key. :

```
// Sortie "Larry"
echo $this->Cookie->read('name');

// Vous pouvez aussi utiliser la notation par point pour lire
echo $this->Cookie->read('User.name');

// Pour prendre les variables que vous aviez groupées en utilisant
// la notation par point comme tableau, faîtes quelque chose comme
$this->Cookie->read('User');

// ceci retourne quelque chose comme array('name' => 'Larry', 'role' => 'Lead')
```

`CookieComponent::check`(*$key*)

Paramètres

- **$key** (*string*) – La clé à vérifier.

Utilisé pour vérifier si une clé/chemin existe et a une valeur non null. Introduit dans la version 2.3 : `CookieComponent::check()` a été ajoutée dans la versoin 2.3

`CookieComponent::delete(mixed $key)`
Efface une variable de cookie du nom défini dans $key. Fonctionne avec la notation par point :

```
// Efface une variable
$this->Cookie->delete('bar');

// Efface la variable bar du cookie, mais seulement dans foo.
$this->Cookie->delete('foo.bar');
```

`CookieComponent::destroy()`
Detruit le cookie actuel.

`CookieComponent::type($type)`
Vous permet de changer le schéma de chiffrement. Par défaut, le schéma 'cipher' est utilisé. Cependant, vous devriez utiliser le schéma 'rijndael' pour une sécurité améliorée. Modifié dans la version 2.2 : Le type 'rijndael' a été ajouté.

EmailComponent

`EmailComponent` est maintenant déprécié, mais il continue à fonctionner. En interne, cette classe utilise `CakeEmail` pour envoyer des mails. Malheureusement, vous aurez besoin de déplacer vos fichiers depuis `app/views/elements/emails` vers `app/View/Emails`. Renommez également le répertoire `email` en `Emails` dans le chemin des layouts. Si cela affecte d'autres endroits dans votre application, nous vous recommandons d'utiliser des liens symboliques.

Nous vous recommandons de mettre à jour votre code pour utiliser la classe `CakeEmail` au lieu de `EmailComponent`. Ci-dessous quelques conseils à propos de la migration.

- L'entête n'est pas changée pour être X-... Ce que vous paramétrez est ce qui est utilisé. Donc rappelez-vous de mettre X- dans vos entêtes perso
- La méthode `send()` ne reçoit uniquement que le contenu du message. Le template et le layout devrons être paramétrés en utilisant la méthode `CakeEmail::template()`. La liste des pièces-jointes devront être un tableau de fichiers (qui apparaîtrons dans l'email) comme clé et valeur le chemin complet du fichier réel.
- A chaque erreur, `CakeEmail` enverra une exception au lieu de retourner false. Nous vous recommandons d'utiliser try/catch pour vous assurer que vos messages sont délivrés correctement.

Ci-dessous quelques exemples d'utilisation du component Email `EmailComponent ($component)` et maintenant avec `CakeEmail ($lib)` :

- De `$component->to = 'quelquechose@exemple.com';` vers `$lib->to('quelquechose@exemple.com');`
- De `$component->to = 'Alias <quelquechose@exemple.com>';` vers `$lib->to('quelquechose@exemple.com', 'Alias');` or `$lib->to(array('quelquechose@exemple.com' => 'Alias'));`
- De `$component->subject = 'Mon sujet';` vers `$lib->subject('Mon sujet');`
- De `$component->date = 'Sun, 25 Apr 2011 01:00:00 -0300';` vers `$lib->addHeaders(array('Date' => 'Sun, 25 Apr 2011 01:00:00 -0300'));`

– De `$component->header['Custom'] = 'only my';` vers `$lib->addHeaders(array('X-Custom' => 'only my'));`
– De `$component->send(null, 'template', 'layout');` vers `$lib->template('template', 'layout')->send();`
– De `$component->delivery = 'smtp';` vers `$lib->transport('Smtp');`
– De `$component->smtpOptions = array('host' => 'smtp.exemple.com');` vers `$lib->config(array('host' => 'smtp.exemple.com'));`
– De `$sent = $component->httpMessage;` vers `$sent = $lib->message(CakeEmail::MESSAGE_HT`

Pour plus d'informations, vous devriez lire la documentation *CakeEmail*.

Request Handling (Gestion des requêtes)

class `RequestHandlerComponent` (*ComponentCollection $collection*, *array $settings = array()*)

Le component Request Handler est utilisé dans CakePHP pour obtenir des informations supplémentaires au sujet des requêtes HTTP qui sont faites à votre application. Vous pouvez l'utiliser pour informer vos controllers des processus Ajax, autant que pour obtenir des informations complémentaires sur les types de contenu que le client accepte et modifie automatiquement dans le layout approprié, quand les extensions de fichier sont disponibles.

Par défaut, le RequestHandler détectera automatiquement les requêtes Ajax basées sur le header HTTP-X-Requested-With, qui est utilisé par de nombreuses librairies javascript. Quand il est utilisé conjointement avec `Router::parseExtensions()`, RequestHandler changera automatiquement le layout et les fichiers de vue par ceux qui correspondent au type demandé. En outre, s'il existe un helper avec le même nom que l'extension demandée, il sera ajouté au tableau des helpers des Controllers. Enfin, si une donnée XML/JSON est POST'ée vers vos Controllers, elle sera décomposée dans un tableau qui est assigné à `$this->request->data`, et pourra alors être sauvegardée comme une donnée de model. Afin d'utiliser le Request Handler, il doit être inclu dans votre tableau $components :

```
class WidgetController extends AppController {

    public $components = array('RequestHandler');

    // suite du controller
}
```

Obtenir des informations sur une requête

Request Handler contient plusieurs méthodes qui fournissent des informations à propos du client et de ses requêtes.

`RequestHandlerComponent::`**`accepts`**(*$type = null*)

$type peut être une chaîne, un tableau, ou 'null'. Si c'est une chaîne, la méthode accepts() renverra true si le client accepte ce type de contenu. Si c'est un tableau, accepts() renverra true si un des types du contenu est accepté par le client. Si c'est 'null', elle renverra un tableau des types de contenu que le client accepte. Par exemple :

```
class PostsController extends AppController {

    public $components = array('RequestHandler');

    public function beforeFilter () {
        if ($this->RequestHandler->accepts('html')) {
            // Ce code est exécuté uniquement si le client accepte les réponses HTML (text/
        } elseif ($this->RequestHandler->accepts('xml')) {
            // exécuté seulement si le client accepte seulement les réponse XML
        }
        if ($this->RequestHandler->accepts(array('xml', 'rss', 'atom'))) {
            // Executé si le client accepte l'un des suivants: XML, RSS ou Atom
        }
    }
}
```

D'autres méthodes de détections du contenu des requêtes :

`RequestHandlerComponent::isXml()`
Renvoie true si la requête actuelle accepte les réponses XML.

`RequestHandlerComponent::isRss()`
Renvoie true si la requête actuelle accepte les réponses RSS.

`RequestHandlerComponent::isAtom()`
Renvoie true si l'appel actuel accepte les réponse Atom, false dans le cas contraire.

`RequestHandlerComponent::isMobile()`
Renvoie true si le navigateur du client correspond à un téléphone portable, ou si le client accepte le contenu WAP. Les navigateurs mobiles supportés sont les suivants :
–Android
–AvantGo
–BlackBerry
–DoCoMo
–Fennec
–iPad
–iPhone
–iPod
–J2ME
–MIDP
–NetFront
–Nokia
–Opera Mini
–Opera Mobi
–PalmOS
–PalmSource
–portalmmm
–Plucker
–ReqwirelessWeb
–SonyEricsson
–Symbian

–UP.Browser
–webOS
–Windows CE
–Windows Phone OS
–Xiino

`RequestHandlerComponent::isWap()`

Retourne true si le client accepte le contenu WAP.

Toutes les méthodes de détection des requêtes précédentes peuvent être utilisées dans un contexte similaire pour filtrer les fonctionnalités destinées à du contenu spécifique. Par exemple, au moment de répondre aux requêtes AJAX, si vous voulez désactiver le cache du navigateur, et changer le niveau de débogage. Cependant, si vous voulez utiliser le cache pour les requêtes non-AJAX., le code suivant vous permettra de le faire :

```
if ($this->request->is('ajax')) {
    $this->disableCache();
}
// Continue l'action du controller
```

Obtenir des informations supplémentaires sur le client

`RequestHandlerComponent::getAjaxVersion()`

Récupère la version de la librairie 'Prototype' si la requête est de type AJAX ou une chaîne de caractères vide dans le cas contraire. La librairie 'Prototype' envoie une entête HTTP spéciale "Prototype version".

Décoder automatiquement les données de la requête

`RequestHandlerComponent::addInputType($type, $handler)`

Paramètres

- **$type** (*string*) – L'alias du type de contenu auquel ce décodeur est attaché. ex. 'json' ou 'xml'
- **$handler** (*array*) – L'information de gestionnaire pour le type.

Ajoute une requête de décodage de données. Le gestionnaire devrait contenir un callback, et tour autre argument supplémentaire pour le callback. Le callback devrait retourner un tableau de données contenues dans la requête. Par exemple, ajouter un gestionnaire de CSV dans le callback 'beforeFilter' de votre controller pourrait ressembler à ceci

```
$parser = function ($data) {
    $rows = str_getcsv($data, "\n");
    foreach ($rows as &$row) {
        $row = str_getcsv($row, ',');
    }
    return $rows;
};
$this->RequestHandler->addInputType('csv', array($parser));
```

L'exemple ci-dessus nécessite PHP 5.3, cependant vous pouvez utiliser n'importe quel callback (http ://php.net/callback) pour la fonction de gestion. Vous pouvez aussi passer des arguments supplémentaires au callback, c'est très utile pour les callbacks comme `json_decode` :

```
$this->RequestHandler->addInputType('json', array('json_decode', true));
```

Le contenu ci-dessus créera `$this->request->data` un tableau des données d'entrées JSON, sans le `true` supplémentaire vous obtiendrez un jeu d'objets `StdClass`.

Répondre Aux Requêtes

En plus de la détection de requêtes, RequestHandler fournit également une solution simple pour modifier la sortie de façon à ce que le type de contenu corresponde à votre application.

`RequestHandlerComponent::setContent`(*$name*, *$type = null*)

Paramètres

- **$name** (*string*) – Le nom ou l'extension du fichier (Content-type), par ex : html, css, json, xml.
- **$type** (*mixed*) –
  - Le(s) type(s) mime(s) auquel se réfère Content-type.

setContent ajoute/définit les Content-types pour le nom précisé. Permet aux content-types d'être associés à des alias simplifiés et/ou à des extensions. Ceci permet à RequestHandler de répondre automatiquement aux requêtes de chaque type dans sa méthode startup. Si vous utilisez Router : :parseExtension, vous devriez utiliser l'extension de fichier comme le nom du Content-type. De plus, ces types de contenu sont utilisés par prefers() et accepts().

setContent est bien mieux utilisé dans le beforeFilter() de vos controllers, parce qu'il tirera un meilleur profit de l'automagie des alias de content-type.

Les correspondances par défaut sont :
–**javascript** text/javascript
–**js** text/javascript
–**json** application/json
–**css** text/css
–**html** text/html, */*
–**text** text/plain
–**txt** text/plain
–**csv** application/vnd.ms-excel, text/plain
–**form** application/x-www-form-urlencoded
–**file** multipart/form-data
–**xhtml** application/xhtml+xml, application/xhtml, text/xhtml
–**xhtml-mobile** application/vnd.wap.xhtml+xml
–**xml** application/xml, text/xml
–**rss** application/rss+xml
–**atom** application/atom+xml
–**amf** application/x-amf
–**wap** text/vnd.wap.wml, text/vnd.wap.wmlscript, image/vnd.wap.wbmp
–**wml** text/vnd.wap.wml
–**wmlscript** text/vnd.wap.wmlscript

–**wbmp** image/vnd.wap.wbmp
–**pdf** application/pdf
–**zip** application/x-zip
–**tar** application/x-tar

`RequestHandlerComponent::prefers`(*$type = null*)

Détermine quels content-types le client préfère. Si aucun paramètre n'est donné, le type de contenu le plus approchant est retourné. Si $type est un tableau, le premier type que le client accepte sera retourné. La préférence est déterminée, premièrement par l'extension de fichier analysée par Router, si il y en avait une de fournie et secondairement, par la liste des content-types définis dans HTTP_ACCEPT.

`RequestHandlerComponent::renderAs`(*$controller*, *$type*)

Paramètres

- **$controller** (*Controller*) – Référence du controller
- **$type** (*string*) – nom simplifié du type de contenu à rendre, par exemple : xml, rss.

Change le mode de rendu d'un controller pour le type spécifié. Ajoutera aussi le helper approprié au tableau des helpers du controller, s'il est disponible et qu'il n'est pas déjà dans le tableau.

`RequestHandlerComponent::respondAs`(*$type*, *$options*)

Paramètres

- **$type** (*string*) – nom simplifié du type de contenu à rendre, par exemple : xml, rss ou un content-type complet, tel que application/x-shockwave
- **$options** (*array*) – Si $type est un nom simplifié de type, qui a plus d'une association avec des contenus, $index est utilisé pour sélectionner le type de contenu.

Définit l'en-tête de réponse basé sur la correspondance content-type/noms.

`RequestHandlerComponent::responseType`()

Retourne l'en-tête Content-type du type de réponse actuel ou null s'il y en a déjà un de défini.

Profiter du cache de validation HTTP

Introduit dans la version 2.1. Le model de validation de cache HTTP est l'un des processus utilisé pour les passerelles de cache, aussi connu comme reverse proxies, pour déterminer si elles peuvent servir une copie de réponse stockée au client. D'après ce model, vous bénéficiez surtout d'une meilleur bande passante, mais utilisé correctement vous pouvez aussi gagner en temps de processeur, et ainsi gagner en temps de réponse.

En activant le Component RequestHandler `RequestHandlerComponent` dans votre controller vous validerez le contrôle automatique effectué avant de rendre une vue. Ce contrôle compare l'objet réponse à la requête originale pour déterminer si la réponse n'a pas été modifiée depuis la dernière fois que le client a fait sa demande.

Si la réponse est évaluée comme non modifiée, alors le processus de rendu de vues est arrêter, réduisant le temps processeur. Un `no content` est retourné au client, augmentant la bande passante. Le code de réponse est défini à *304 Not Modified*.

Vous pouvez mettre en retrait ce contrôle automatique en paramétrant `checkHttpCache` à false :

```
public $components = array(
    'RequestHandler' => array(
        'checkHttpCache' => false
));
```

Utiliser les ViewClasses personnalisées

Introduit dans la version 2.3. Quand vous utilisez JsonView/XmlView, vous aurez envie peut-être de surcharger la serialization par défaut avec une classe View par défaut, ou ajouter des classes View pour d'autres types.

Vous pouvez mapper les types existants et les nouveaux types à vos classes personnalisées.

`RequestHandlerComponent::viewClassMap`(*$type*, *$viewClass*)

Paramètres

- **$type** (*string|array*) – Le type string ou un tableau map avec le format `array('json' => 'MyJson')`.
- **$viewClass** (*string*) – La viewClass à utiliser pour le type sans *View* en suffixe.

Vous pouvez aussi définir ceci automatiquement en utilisant la configuration `viewClassMap` :

```
public $components = array(
    'RequestHandler' => array(
        'viewClassMap' => array(
            'json' => 'ApiKit.MyJson',
            'xml' => 'ApiKit.MyXml',
            'csv' => 'ApiKit.Csv'
        )
));
```

Pagination

class `PaginatorComponent`(*ComponentCollection $collection*, *array $settings = array()*)

Un des principaux obstacles à la création d'une application flexible et ergonomique est le design et une interface utilisateur intuitive. De nombreuses applications ont tendance à augmenter en taille et en complexité rapidement, et les designers ainsi que les programmeurs trouvent même qu'ils sont incapables de faire face a l'affichage des centaines ou des milliers d'enregistrements. Réécrire prend du temps, et les performances et la satisfaction des utilisateurs peut en pâtir.

Afficher un nombre raisonnable d'enregistrements par page a toujours été une partie critique dans toutes les applications et cause régulièrement de nombreux maux de tête aux développeurs. CakePHP allège le fardeau des développeurs en fournissant un moyen rapide et facile de paginer les données.

La pagination dans CakePHP est offerte par un Component dans le controller, pour rendre la création des requêtes de pagination plus facile. Dans la Vue, `PaginatorHelper` est utilisé pour rendre la génération de la pagination, des liens et des boutons simples.

Paramétrage des requêtes

Dans le controller, nous commençons par définir les conditions de la requête de pagination qui seront utilisées par défaut dans la variable `$paginate` du controller. Ces conditions, vont servir de base à vos requêtes de pagination. Elles sont complétées par le tri, la direction, la limitation et les paramètres de page passés depuis l'url. Ici, il est important de noter que l'ordre des clés doit être défini dans une structure en tableau comme ci-dessous :

```
class PostsController extends AppController {

    public $paginate = array(
        'limit' => 25,
        'order' => array(
            'Post.title' => 'asc'
        )
    );
}
```

Vous pouvez aussi inclure d'autres options `find()`, comme `fields` :

```
class PostsController extends AppController {

    public $paginate = array(
        'fields' => array('Post.id', 'Post.created'),
        'limit' => 25,
        'order' => array(
            'Post.title' => 'asc'
        )
    );
}
```

D'autres clés qui peuvent être introduites dans le tableau `$paginate` sont similaires aux paramètres de la méthode `Model->find('all')`, qui sont : `conditions`, `fields`, `order`, `limit`, `page`, `contain`,''joins'', et `recursive`. En plus des touches mentionnées ci dessus, chacune des clés peut aussi être passé à la méthode find du model. Ça devient alors très simple d'utiliser les component comme `ContainableBehavior` avec la pagination :

```
class RecipesController extends AppController {

    public $paginate = array(
        'limit' => 25,
        'contain' => array('Article')
    );
}
```

En plus de définir des valeurs de pagination générales, vous pouvez définir plus d'un jeu de pagination par défaut dans votre controller, vous avez juste à nommer les clés du tableau d'après le model que vous souhaitez configurer :

```
class PostsController extends AppController {

    public $paginate = array(
```

```
        'Post' => array (...),
        'Author' => array (...)
    );
}
```

Les valeurs des clés `Post` et `Author` pourraient contenir toutes les propriétés qu'un model/clé sans `$paginate` pourraient contenir.

Une fois que la variable `$paginate` à été définie, nous pouvons appeler la méthode `paginate()` dans l'action du controller. Cette méthode chargera dynamiquement `PaginatorComponent`, et appellera sa méthode paginate(). Ceci retournera le résultat du `find()` depuis le model. Ceci envoi également quelques statistiques de pagination, qui sont additionnées à l'objet de la requête. L'information supplémentaire est envoyée à `$this->request->params['paging']`, et est utilisée par `PaginatorHelper` pour la création de liens. `Controller::paginate()` ajoute également PaginatorHelper à la liste des helpers de votre controller, si il n'a pas encore été additionné :

```
public function list_recipes() {
    // similaire à un  findAll(), mais récupère les résultats paginés
    $data = $this->paginate('Recipe');
    $this->set('data', $data);
}
```

Vous pouvez filtrer les enregistrements en passant des conditions en second paramètre à la fonction `paginate()`.:

```
$data = $this->paginate('Recipe', array('Recipe.title LIKE' => 'a%'));
```

Ou vous pouvez aussi définir des `conditions` et d'autres clés dans le tableau `$paginate` à l'intérieur de votre action. :

```
public function list_recipes() {
    $this->paginate = array(
        'conditions' => array('Recipe.title LIKE' => 'a%'),
        'limit' => 10
    );
    $data = $this->paginate('Recipe');
    $this->set(compact('data'));
);
```

Personnalisation des requêtes de pagination

Si vous n'êtes pas prêts à utiliser les options standards du find pour créer la requête d'affichage de vos données, il y a quelques options. Vous pouvez utiliser *custom find type*. Vous pouvez aussi implémenter les méthodes `paginate()` et `paginateCount()` sur votre model, ou les inclure dans un behavior attaché à votre model. Les behaviors qui implémentent `paginate` et/ou `paginateCount` devraient implémenter les signatures de méthode définies ci-dessous avec le premier paramètre normal supplémentaire de `$model` :

```
// paginate et paginateCount implémentés dans le behavior.
public function paginate(Model $model, $conditions, $fields, $order, $limit, $page = 1, $re
    // contenu de la méthode
```

```
}

public function paginateCount(Model $model, $conditions = null, $recursive = 0, $extra = ar
    // corps (body) de la méthode
}
```

C'est rare d'avoir besoin d'implémenter paginate() et paginateCount(). vous devriez vous assurer que vous ne pouvez pas atteindre votre but avec les méthodes du noyau du model, ou avec un finder personnalisé. Pour paginer avec un type de find personnalisé, vous devez définir le `0`'ème element, ou la clé `findType` depuis la version 2.3 :

```
public $paginate = array(
    'popular'
);
```

Puisque le 0ème index est difficile à gérer, dans 2.3 l'option `findType` a été ajoutée :

```
public $paginate = array(
    'findType' => 'popular'
);
```

La méthode `paginate()` devrait implémenter les signatures de méthode suivantes. Pour utiliser vos propres méthodes/logiques, surchargez les dans le model dans lequel vous voulez récupérer des données :

```
/**
 * Surcharge de la méthode paginate - groupée par week, away_team_id et home_team_id
 */
public function paginate($conditions, $fields, $order, $limit, $page = 1, $recursive = null
    $recursive = -1;
    $group = $fields = array('week', 'away_team_id', 'home_team_id');
     return $this->find('all', compact('conditions', 'fields', 'order', 'limit', 'page', '
}
```

Vous aurez aussi besoin de surcharger le `paginateCount()` du noyau, cette méthode s'attend aux mêmes arguments que `Model::find('count')`. L'exemple ci-dessous utilise quelques fonctionnalités Postgres spécifiques, Veuillez ajuster en conséquence en fonction de la base de données que vous utilisez :

```
/**
 * Surcharge de la méthode paginateCount
 */
public function paginateCount($conditions = null, $recursive = 0, $extra = array()) {
    $sql = "SELECT DISTINCT ON(week, home_team_id, away_team_id) week, home_team_id, away_t
    $this->recursive = $recursive;
    $results = $this->query($sql);
    return count($results);
}
```

Le lecteur attentif aura noté que la méthode paginate que nous avons définie n'était pas réellement nécessaire - Tout ce que vous avez à faire est d'ajouter le mot clé dans la variable de classe `$paginate` du controller :

```
/**
 * Ajout d'une clause GROUP BY
 */
public $paginate = array(
    'MyModel' => array(
        'limit' => 20,
        'order' => array('week' => 'desc'),
        'group' => array('week', 'home_team_id', 'away_team_id')
    )
);
/**
 * Ou à la volée depuis l'intérieur de l'action
 */
public function index() {
    $this->paginate = array(
        'MyModel' => array(
            'limit' => 20,
            'order' => array('week' => 'desc'),
            'group' => array('week', 'home_team_id', 'away_team_id')
        )
    );
```

Dans CakePHP 2.0, vous n'avez plus besoin d'implémenter `paginateCount()` quand vous utilisez des clauses de groupe. Le `find('count')` du groupe comptera correctement le nombre total de lignes.

Contrôle du champ à utiliser pour ordonner

Par défaut le classement peut être effectué pour n'importe quelle colonne dans un model. C'est parfois indésirable comme permettre aux utilisateurs de trier des colonnes non indexées, ou des champs virtuels ce qui peut être coûteux en temps de calculs. Vous pouvez utiliser le 3ème paramètre de `Controller::paginate()` pour restreindre les tris de colonnes qui pourront être effectués :

```
$this->paginate('Post', array(), array('title', 'slug'));
```

Ceci permettrait le tri uniquement sur les colonnes title et slug. Un utilisateur qui paramètre le tri à d'autres valeurs sera ignoré.

Limitation du nombre maximum de lignes qui peuvent être recherchées

Le nombre de résultats qui sont retournés à l'utilisateur est représenté par le paramètre `limit`. Il est généralement indésirable de permettre à l'utilisateur de retourner toutes les lignes dans un ensemble paginé. Par défaut CAKEPHP limite le nombre de lignes retournées à 100. Si cette valeur par défaut n'est pas appropriée pour votre application, vous pouvez l'ajuster dans une partie des options de pagination :

```
public $paginate = array(
    // d'autre clés ici.
    'maxLimit' => 10
);
```

Si le paramètre de limitation de la requête est supérieur à cette valeur, il sera réduit à la valeur de `maxLimit`.

Pagination avec des paramètres GET

Dans les versions précédentes de CAKEPHP vous ne pouviez générer des liens de pagination qu'en utilisant des paramètres nommés. Mais si les pages étaient recherchées avec des paramètres GET elle continueraient à fonctionner. Pour la version 2.0, nous avons décidés de rendre la façon de générer les paramètres de pagination plus contrôlable et plus cohérente. Vous pouvez choisir d'utiliser une chaîne de requête ou bien des paramètre nommés dans le component. Les requêtes entrantes devront accepter le type choisi, et `PaginatorHelper` générera les liens avec les paramètres choisis :

```
public $paginate = array(
    'paramType' => 'querystring'
);
```

Ce qui est au-dessus permet à un paramètre de recherche sous forme de chaîne de caractères, d'être parsé et d'être généré. Vous pouvez aussi modifier les propriétés de `$settings` du Component Paginator (PaginatorComponent) :

```
$this->Paginator->settings['paramType'] = 'querystring';
```

Par défaut tous les paramètres de pagination typiques seront convertis en arguments GET.

Note : Vous pouvez rentrer dans une situation où assigner une valeur dans une propriété inexistante retournera des erreurs :

```
$this->paginate['limit'] = 10;
```

Retournera l'erreur "Notice : Indirect modification of overloaded property $paginate has no effect" ("Notice : Une modification indirect d'une surcharge de la propriété $paginate n'a aucun effet"). En assignant une valeur initiale à la propriété, cela résout le problème :

```
$this->paginate = array();
$this->paginate['limit'] = 10;
//ou
$this->paginate = array('limit' => 10);
```

Ou juste en déclarant la propriété dans la classe du controller

```
class PostsController {
    public $paginate = array();
}
```

Ou en utilisant `$this->Paginator->setting = array('limit' => 10);`

Assurez-vous d'avoir ajouté le component Paginator dans votre tableau $components si vous voulez modifier la propriété `$settings` du Component Paginator.

L'une ou l'autre de ces approches résoudra les erreurs rencontrés.

Requêtes en dehors des clous

Depuis la version 2.3, PaginatorComponent va lancer une *NotFoundException* quand il essaiera d'accéder à une page qui n'existe pas, par ex le nombre de la page requêtée est plus grand que le total du nombre de pages.

Ainsi vous pouvez soit laisser la page d'erreur normal être rendu ou bien vous pouvez utiliser un block try catch et renvoyer vers l'action appropriée quand une exception *NotFoundException* est attrapée :

```
public function index() {
    try {
        $this->paginate();
    } catch (NotFoundException $e) {
        //Faire quelque chose ici comme rediriger à la première ou dernière page.
        //$this->request->params['paging'] va vous donner une l'info nécessaire.
    }
}
```

Pagination AJAX

C'est très simple d'incorporer les fonctionnalités Ajax dans la pagination. en utilisant `JsHelper` et `RequestHandlerComponent` vous pouvez facilement ajouter des paginations Ajax à votre application. Voir *La Pagination Ajax* pour plus d'information.

Pagination dans la vue

Regardez la documentation du `PaginatorHelper` pour voir comment créer des liens de navigation paginés.

Security (Sécurité)

class `SecurityComponent` (*ComponentCollection $collection*, *array $settings = array()*)

Le component Security offre une manière simple d'inclure une sécurité renforcée à votre application. Il fournit des méthodes pour diverses tâches comme :
- Restreindre les méthodes HTTP que votre application accepte.
- Protection CSRF.
- Protection contre la falsification de formulaire.
- Exiger l'utilisation du SSL.
- Limiter les communications croisées dans le controller.

Comme tous les components, il est configuré au travers de plusieurs paramètres configurables. Toutes ces propriétés peuvent être définies directement ou au travers de "méthodes setter" du même nom dans la partie beforeFilter de votre controller.

En utilisant le Component Security vous faites automatiquement un get CSRF (http ://en.wikipedia.org/wiki/Cross-site_request_forgery) et une protection contre la falsification de formulaire. Des jetons de champs cachés seront automatiquement insérés dans les formulaires et vérifiés

par le component Security. En outre, une soumission par formulaire ne sera pas acceptée après une certaine période d'inactivité, qui est contrôler par le temps `csrfExpires`.

Si vous utilisez la fonctionnalité de protection des formulaires par le component Security et que d'autre components traitent des données de formulaire dans les callbacks `startup()`, assurez-vous de placer le component Security avant ces components dans le tableau `$components`.

Note : Quand vous utilisez le component Security vous **devez** utiliser le Helper Form (FormHelper) pour créer vos formulaires. Le component Security regarde certains indicateurs qui sont créés et gérés par le Helper form. (spécialement ceux créés dans `create()`) et `end()`). La modification dynamique des champs qui lui sont soumis dans une requête POST (ex. désactiver, effacer, créer des nouveaux champs via Javascript) est susceptible de déclencher un black-holing (envoi dans le trou noir) de la requête. Voir les paramètres de configuration de `$validatePost` ou `$disabledFields`.

Gestion des callbacks trou noir

Si une action est restreinte par le component Security, elle devient un trou noir, comme une requête invalide qui aboutira à une erreur 404 par défaut. Vous pouvez configurer ce comportement, en définissant la propriété $this->Security->blackHoleCallback par une fonction de rappel (callback) dans le controller.

`SecurityComponent::`**`blackHole`**(*object $controller*, *string $error*)

Met en "trou noir" (black-hole) une requête invalide, avec une erreur 404 ou un callback personnalisé. Sans callback, la requête sera abandonnée. Si un callback de controller est défini pour SecurityComponent : :blackHoleCallback, il sera appelé et passera toute information sur l'erreur.

property `SecurityComponent::$`**`blackHoleCallback`**

La fonction de rappel (callback) du controller qui va gérer et requéter ce qui doit être mis dans un trou noir (blackholed). La fonction de rappel de mise en trou noir (blackhole callback) peut être n'importe quelle méthode publique d'un controller. La fonction de rappel doit s'attendre a un paramètre indiquant le type d'erreur :

```
public function beforeFilter() {
    $this->Security->blackHoleCallback = 'blackhole';
}

public function blackhole($type) {
    // gestions des erreurs.
}
```

Le paramètre `$type` peut avoir les valeurs suivantes :
–'auth' Indique une erreur de validation de formulaire, ou une incohérence controller/action.
–'csrf' Indique une erreur CSRF.
–'get' Indique un problème sur la méthode de restriction HTTP.
–'post' Indique un problème sur la méthode de restriction HTTP.
–'put' Indique un problème sur la méthode de restriction HTTP.
–'delete' Indique un problème sur la méthode de restriction HTTP.
–'secure' Indique un problème sur la méthode de restriction SSL.

Restreindre les méthodes HTTP

`SecurityComponent::requirePost()`
Définit les actions qui nécessitent une requête POST. Prend un nombre indéfini de paramètres. Peut être appelé sans argument, pour forcer toutes les actions à requérir un POST.

`SecurityComponent::requireGet()`
Définit les actions qui nécessitent une requête GET. Prend un nombre indéfini de paramètres. Peut-être appelé sans argument, pour forcer toutes les actions à requérir un GET.

`SecurityComponent::requirePut()`
Définit les actions qui nécessitent une requête PUT. Prend un nombre indéfini de paramètres. Peut-être appelé sans argument, pour forcer toutes les actions à requérir un PUT.

`SecurityComponent::requireDelete()`
Définit les actions qui nécessitent une requête DELETE. Prend un nombre indéfini de paramètres. Peut-être appelé sans argument, pour forcer toutes les actions à requérir un DELETE.

Restreindre les actions à SSL

`SecurityComponent::requireSecure()`
Définit les actions qui nécessitent une requête SSL-securisée. Prend un nombre indéfini de paramètres. Peut-être appelé sans argument, pour forcer toutes les actions à requérir une SSL-securisée.

`SecurityComponent::requireAuth()`
Définit les actions qui nécessitent un jeton valide généré par le component Security. Prend un nombre indéfini de paramètres. Peut-être appelé sans argument, pour forcer toutes les actions à requérir une authentification valide.

Restreindre les demandes croisées de controller

property `SecurityComponent::$allowedControllers`
Une liste de controllers à partir desquelles les actions du controller courant sont autorisées à recevoir des requêtes. Ceci peut être utilisé pour contrôler les demandes croisées de controller.

property `SecurityComponent::$allowedActions`
Les actions parmi celles du controller courant qui sont autorisées à recevoir des requêtes. Ceci peut être utilisé pour contrôler les demandes croisées de controller.

Prévention de la falsification de formulaire

Par défaut le component Security `SecurityComponent` prévient les utilisateurs de la falsification de formulaire. Il fait cela en travaillant avec le Helper Form et en traquant quels fichiers sont dans un formulaire. il assure le suivi des éléments d'entrée cachés. Toutes ces données sont combinées et hachées. Quand un formulaire est soumis, le component de sécurité utilisera les données POSTé pour construire la même structure et comparer le hachage.

property `SecurityComponent::$unlockedFields`
Définit une liste de champs de formulaire à exclure de la validation POST. Les champs peuvent être déverrouillés dans le component ou avec `FormHelper::unlockField()`. Les champs qui ont été déverrouillés ne sont pas requis faisant parti du POST et les champs cachés déverrouillés n'ont pas leur valeur vérifiée.

property `SecurityComponent::$validatePost`
Mis à `false` pour complètement éviter la validation des requêtes POST, essentiellement éteindre la validation de formulaire.

configuration CSRF (Cross site request forgery)

property `SecurityComponent::$csrfCheck`
Si vous utilisez les formulaires de protection CSRF. Définit à `false` pour désactiver la protection CSRF sur les formulaires.

property `SecurityComponent::$csrfExpires`
La durée avant expiration d'un jeton CSRF. Chaque requête formulaire/page va générer un nouveau jeton qui ne pourra être soumis qu'une seule fois avant son expiration. Peut être une valeur compatible avec `strtotime()`. Par défaut 30 minutes.

property `SecurityComponent::$csrfUseOnce`
Contrôle si oui ou non les jetons CSRF sont utilisés et brûlés. Définit à `false` pour ne pas générer de nouveau jetons sur chaque requête. Un jeton pourra être réutilisé jusqu'à ce qu'il expire. Ceci réduit les chances des utilisateurs d'avoir des requêtes invalides en raison de la consommation de jeton. Cela à pour effet de rendre CSRF moins sécurisé, et les jetons réutilisables.

Utilisation

Le component Security est généralement utilisé dans la méthode beforeFilter() de votre controller. Vous pouvez spécifier les restrictions de sécurité que vous voulez et le component Security les forcera au démarrage :

```
class WidgetController extends AppController {

    public $components = array('Security');

    public function beforeFilter() {
        $this->Security->requirePost('delete');
    }
}
```

Dans cette exemple, l'action delete peut être effectuée avec succès si celui ci reçoit une requête POST :

```
class WidgetController extends AppController {

    public $components = array('Security');

    public function beforeFilter() {
```

```
        if (isset($this->request->params['admin'])) {
            $this->Security->requireSecure();
        }
    }
}
```

Cette exemple forcera toutes les actions qui proviennent de la "route" Admin à être effectuées via des requêtes sécurisées SSL :

```
class WidgetController extends AppController {

    public $components = array('Security');

    public function beforeFilter() {
        if (isset($this->params['admin'])) {
            $this->Security->blackHoleCallback = 'forceSSL';
            $this->Security->requireSecure();
        }
    }

    public function forceSSL() {
        $this->redirect('https://' . env('SERVER_NAME') . $this->here);
    }
}
```

Cet exemple forcera toutes les actions qui proviennent de la "route" admin à requérir des requêtes sécurisés SSL. Quand la requête est placée dans un trou noir, elle appellera le callback forceSSL() qui redirigera les requêtes non sécurisées vers les requêtes sécurisées automatiquement.

protection CSRF

CSRF ou Cross Site Request Forgery est une vulnérabilité courante pour les applications Web. Cela permet à un attaquant de capturer et de rejouer une requête, et parfois de soumettre des demandes de données en utilisant les balises images ou des ressources sur d'autres domaines.

Les doubles soumissions et les attaques *replay* sont gérées par les fonctionnalités CSRF du component Security. Elles fonctionnent en ajoutant un jeton spécial pour chaque requête de formulaire. Ce jeton utilisé qu'une fois ne peut pas être utilisé à nouveau. Si une tentative est faîte pour ré-utiliser un jeton expiré la requête sera mise dans le trou noir (blackholed)

Utilisation de la protection CSRF En ajoutant simplement la `SecurityComponent` à votre tableau de component, vous pouvez bénéficier de la protection CSRF fournie. Par défaut les jetons CSRF sont valides 30 minutes et expire à l'utilisation. Vous pouvez contrôler la durée des jetons en paramétrant csrfExpires dans le component. :

```
public $components = array(
    'Security' => array(
        'csrfExpires' => '+1 hour'
    )
);
```

Vous pouvez aussi définir cette propriété dans la partie `beforeFilter` de votre controller. :

```
public function beforeFilter() {
    $this->Security->csrfExpires = '+1 hour';
    // ...
}
```

La valeur de la propriété csrfExpires peut être n'importe quelle valeur compatible à la propriété strtotime() (http ://php.net/manual/en/function.strtotime.php). Par défaut le Helper Form `FormHelper` ajoutera une `data[_Token][key]` contenant le jeton CSRF pour tous les formulaires quand le component est activé.

Gérer les jetons manquants ou périmés Les jetons manquants ou périmés sont gérés de la même façon que d'autres violations de sécurité. Le blackHoleCallback du component Security sera appelé avec un paramètre 'csrf'. Ceci vous aide à filtrer en sortie les problèmes de jeton CSRF, des autres erreurs.

Utilisation de jeton par-session au lieu de jeton à usage unique Par défaut un nouveau jeton est généré à chaque requête, et chaque jeton ne peut être utilisé qu'une seule fois. Si un jeton est utilisé une nouvelle fois, il sera mis dans le trou noir. Parfois , ce comportement est indésirable, et peut créer des problèmes avec les applications "une page". Vous pouvez activer la multi-utilisation des jetons en paramétrant `csrfUseOnce` à `false`. Ceci peut être effectué dans le tableau components, ou dans la partie `beforeFilter` de votre controller :

```
public $components = array(
    'Security' => array(
        'csrfUseOnce' => false
    )
);
```

Cela dira au component que vous voulez ré-utiliser un jeton CSRF jusqu'à ce qu'il expire - C'est contrôlé par les valeurs de `csrfExpires`. Si vous avez des problèmes avec les jetons expirés, ceci peut être une bon équilibrage entre la sécurité et la facilité d'utilisation.

Désactiver la protection CSRF Il peut y avoir des cas où vous souhaitez désactiver la protection CSRF sur vos formulaires. Si vous voulez désactiver cette fonctionnalité, vous pouvez définir `$this->Security->csrfCheck = false;` dans votre `beforeFilter` ou utiliser le tableau components. Par défaut la protection CSRF est activée, et paramétrée pour l'utilisation de jetons à usage unique.

Désactiver le Component Security pour des Actions Spécifiques

Il peut arriver que vous souhaitiez désactiver toutes les vérifications de sécurité pour une action (ex. ajax request). Vous pouvez "délocker" ces actions en les listant dans `$this->Security->unlockedActions` dans votre `beforeFilter`. Introduit dans la version 2.3.

Sessions

class SessionComponent (*ComponentCollection $collection*, *array $settings = array()*)

Le component session de CakePHP fournit le moyen de faire persister les données client entre les pages requêtées. Il agit comme une interface pour $_SESSION et offre aussi des méthodes pratiques pour de nombreuses fonctions relatives à $_SESSION.

Les sessions peuvent être paramétrées de différentes façons dans CakePHP. Pour plus d'information, vous devriez lire la documentation *Session configuration*

Interagir avec les données de Session

Le component Session est utilisé pour interagir avec les informations de session. Il inclut les fonctions CRUD basiques, mais aussi des fonctionnalités pour créer des messages de feedback aux utilisateurs.

Il est important de noter que ces structures en tableaux peuvent être créées dans la session en utilisant la notation avec un point *dot notation*. Par exemple, `User.username` se référera au tableau suivant

```
array('User' =>
    array('username' => 'clark-kent@dailyplanet.com')
);
```

Les points sont utilisés pour indiquer les tableaux imbriqués. Cette notation est utilisée pour toutes les méthodes du component Session quelques soient le nom/la clé utilisé.

`SessionComponent::write`(*$name*, *$value*)

Écrit dans la Session, en mettant $value dans $name. $name peut-être un tableau séparé par un point. Par exemple

```
$this->Session->write('Person.eyeColor', 'Green');
```

Cela écrit la valeur 'Green' dans la session sous Person => eyeColor.

`SessionComponent::read`(*$name*)

Retourne la valeur de $name dans la Session. Si $name vaut null, la session entière sera retournée. Par ex

```
$green = $this->Session->read('Person.eyeColor');
```

Récupère la valeur "Green" de la session. La lecture de données inexistante retournera null.

`SessionComponent::check`(*$name*)

Utilisée pour vérifier qu'une variable de Session a été créée. Retourne true si la variable existe et false dans le cas contraire.

`SessionComponent::delete`(*$name*)

Supprime les données de Session de $name. Par ex

$this->Session->delete('Person.eyeColor');

Notre donnée de session n'a plus la valeur 'Green' ni même l'index eyeColor attribué. Cependant, le model Person est toujours dans la Session. Pour supprimer de la session toutes les informations de Person, utilisez

```
$this->Session->delete('Person');
```

`SessionComponent::destroy()`

La méthode `destroy` supprimera le cookie de session et toutes les données de session stockées dans le fichier temporaire du système. Cela va détruire la session PHP et ainsi en créer une nouvelle. :

```
$this->Session->destroy();
```

Création de messages de notification

`SessionComponent::setFlash(string $message, string $element = 'default', array $params = array(), string $key = 'flash')`

Type retourné void

Souvent dans les applications web, vous aurez besoin d'afficher des messages de notification instantanés à l'utilisateur après avoir terminer un processus ou une réception de données. Dans CakePHP, ceci est appelé "messages flash". Vous pouvez définir des messages flash avec le component Session et les afficher avec le helper session `SessionHelper::flash()`. Pour définir un message, utilisez `setFlash` :

```
// Dans le controller.
$this->Session->setFlash('Votre travail a été sauvegardé !');
```

Ceci créera un message instantané qui peut être affiché à l'utilisateur, en utilisant le Helper Session SessionHelper :

```
// Dans la vue.
echo $this->Session->flash();

// Ce qui générera en sortie.
<div id="flashMessage" class="message">
    Votre travail a été sauvegardé !
</div>
```

Vous pouvez utiliser des paramètres supplémentaires de `setFlash()` pour créer différentes sortes de messages flash. Par exemple, les erreurs et les notifications positives peuvent avoir des apparences différentes. CakePHP vous donne un moyen de le faire. En utilisant le paramètre `$key` vous pouvez stocker différents messages, qui peuvent être séparément récupérer en sortie. :

```
// définit le message que ca va mal
$this->Session->setFlash('Ca va mal.', 'default', array(), 'mal');

// définit le message que ca va bien
$this->Session->setFlash('Ca va bien', 'default', array(), 'bien');
```

Dans la vue, ces messages peuvent être ressortis et stylisés différemment :

```
// dans la vue.
echo $this->Session->flash('bien');
echo $this->Session->flash('mal');
```

Le paramètre `$element` vous permet de contrôler quel élément (localisé dans `/app/View/Elements`) devra être utilisé pour rendre le message. Dans l'élément le message est disponible en tant que `$message`. D'abord nous paramétrons le flash dans notre controller :

```
$this->Session->setFlash('truc customisés', 'flash_custom');
```

Ensuite nous créons le fichier `app/View/Elements/flash_custom.ctp` et créons notre élément flash personnalisé :

```
<div id="myCustomFlash"><?php echo $message; ?></div>
```

`$params` vous permet de passer des variables de vue supplémentaires au layout de rendu. Les paramètres peuvent être passés en affectant la div de rendu, par exemple en ajoutant "class" dans le tableau $params qui appliquera une classe à la div de sortie en utilisant `$this->Session->flash()` dans votre layout ou vue. :

```
$this->Session->setFlash('Message Exemple', 'default', array('class' => 'classe_exempl
```

La sortie en utilisant `$this->Session->flash()` avec l'exemple ci-dessus sera :

```
<div id="flashMessage" class="classe_exemple">Message Exemple</div>
```

Pour utiliser un élément depuis un plugin spécifiez le plugin dans le `$params` :

```
// Utilisera  /app/Plugin/Comment/View/Elements/flash_no_spam.ctp
$this->Session->setFlash('Message!', 'flash_no_spam', array('plugin' => 'Comment'));
```

Helpers (Assistants)

CakePHP dispose d'un certain nombre de helpers qui aident à la création de vues. Ils aident à la création d'un balisage bien formé (y compris les formulaires), l'aide au formatage du texte, des temps et des nombres, et peut même intégrer des bibliothèques javascript populaires. Voici un résumé des helpers intégrés.

Lire *Helpers (Assistants)* pour en apprendre plus sur les helpers, leur api, et comment vous pouvez créer et utiliser vos propres helpers.

Helpers (Assistants)

CacheHelper

class `CacheHelper` (*View $view*, *array $settings = array()*)

Le helper Cache permet la mise en cache des layouts (mises en page) et des vues permettant de gagner du temps pour la récupération de données répétitives. Le système de cache des vues de Cake parse les layout et les vues comme de simple fichier PHP + HTML. Il faut noter que le helper Cache fonctionne de façon assez différente des autres helpers. Il ne possède pas de méthodes appelées directement. A la place, une vue est marquée de tags, indiquant quels blocs de contenus ne doivent pas être mis en cache. Le Helper Cache utilise alors les callbacks du helper pour traiter le fichier et ressortir pour générer le fichier de cache.

Quand une URL est appelée, Cake vérifie si cette requête a déjà été mise en cache. Si c'est le cas, le processus de distribution de l'URL est abandonné. Chacun des blocs non mis en cache sont rendus selon le processus

normal, et la vue est servie. Cela permet de gagner beaucoup de temps pour chaque requête vers une URL mise en cache, puisqu'un minimum de code est exécuté. Si Cake ne trouve pas une vue mise en cache, ou si le cache a expiré pour l'URL appelée, le processus de requête normal se poursuit.

Utilisation du Helper

Il y a deux étapes à franchir avant de pouvoir utiliser le Helper Cache. Premièrement dans votre `APP/Config/core.php` dé-commenter l'appel Configure write pour `Cache.check`. Ceci dira à CakePHP de regarder dans le cache , et de générer l'affichage des fichiers en cache lors du traitement des demandes.

Une fois que vous avez décommenté la ligne `Cache.check` vous devez ajouter le helper à votre tableau `$helpers` de votre controller :

```
class PostsController extends AppController {
    public $helpers = array('Cache');
}
```

Vous devrez aussi ajouter CacheDispatcher à vos filtres de dispatcher dans votre bootstrap :

```
Configure::write('Dispatcher.filters', array(
    'CacheDispatcher'
));
```

Introduit dans la version 2.3 : Si vous avez une configuration avec des domaines ou des languages multiples, vous pouvez utiliser *Configure : :write('Cache.viewPrefix', 'YOURPREFIX') ;* pour stocker les fichiers de vue préfixés mis en cache.

Options de configuration supplémentaires Le Helper Cache (CacheHelper) dispose de plusieurs options de configuration additionnelles que vous pouvez utiliser pour ajuster et régler ces comportements. Ceci est réalisé a travers la variable `$cacheAction` dans vos controllers. `$cacheAction` doit être régler par un tableau qui contient l'action que vous voulez cacher, et la durée en seconde durant laquelle vous voulez que cette vue soit cachée. La valeur du temps peut être exprimé dans le format `strtotime()`. (ex. "1 hour", ou "3 minutes").

En utilisant l'exemple d'un controller d'articles ArticlesController, qui reçoit beaucoup de trafics qui ont besoins d'être mise en cache :

```
public $cacheAction = array(
    'view' => 36000,
    'index'  => 48000
);
```

Ceci mettra en cache l'action view 10 heures et l'action index 13 heures. En plaçant une valeur usuelle de `strtotime()` dans `$cacheAction` vous pouvez mettre en cache toutes les actions dans le controller :

```
public $cacheAction = "1 hour";
```

Vous pouvez aussi activer les callbacks controller/component pour les vues cachées créées avec `CacheHelper`. Pour faire cela, vous devez utiliser le format de tableau pour `$cacheAction` et créer un tableau comme ceci :

```
public $cacheAction = array(
    'view' => array('callbacks' => true, 'duration' => 21600),
    'add' => array('callbacks' => true, 'duration' => 36000),
    'index' => array('callbacks' => true, 'duration' => 48000)
);
```

En paramétrant `callbacks => true` vous dites au CacheHelper (Assistant Cache) que vous voulez que les fichiers générés créent les components et les models pour le controller. De manière additionnelle, lance la méthode initialize du component, le beforeFilter du controller , et le démarrage des callbacks de component.

Note : Définir callbacks => true fait échouer en partie le but de la mise en cache. C'est aussi la raison pour laquelle ceci est désactivé par défaut.

Marquer les contenus Non-Cachés dans les Vues

Il y aura des fois où vous ne voudrez par mettre en cache une vue *intégrale*. Par exemple, certaines parties d'une page peuvent être différentes, selon que l'utilisateur est actuellement identifié ou qu'il visite votre site en tant qu'invité.

Pour indiquer que des blocs de contenu *ne doivent pas* être mis en cache, entourez-les par < !–nocache–> < !–/nocache–>'' comme ci-dessous :

```
<!--nocache-->
<?php if ($this->Session->check('User.name')) : ?>
    Bienvenue, <?php echo h($this->Session->read('User.name')); ?>.
<?php else: ?>
    <?php echo $html->link('Login', 'users/login')?>
<?php endif; ?>
<!--/nocache-->
```

Note : Vous ne pouvez pas utilisez les tags `nocache` dans les éléments. puisqu'il n'y a pas de callbacks autour des éléments, ils ne peuvent être cachés.

Il est à noter, qu'une fois une action mise en cache, la méthode du controller correspondante ne sera plus appelée. Quand un fichier cache est créé, l'objet request, et les variables de vues sont sérialisées avec `serialize()` de PHP.

> **Warning :** Si vous avez des variables de vues qui contiennent des contenus inserialisable comme les objets SimpleXML, des gestionnaires de ressource (resource handles), ou des classes closures Il se peut que vous ne puissiez pas utiliser la mise en cache des vues.

Nettoyer le Cache

Il est important de se rappeler que Cake va nettoyer le cache si un model utilisé dans la vue mise en cache a été modifié. Par exemple, si une vue mise en cache utilise des données du model Post et qu'il y a eu une requête INSERT, UPDATE, ou DELETE sur Post, le cache de cette vue est nettoyé, et un nouveau contenu sera généré à la prochaine requête.

Note : Ce système de nettoyage automatique requiert que le nom du controller/model fasse partie de l'Url. Si vous avez utilisé le routing pour changer vos Urls cela ne fonctionnera pas.

Si vous avez besoin de nettoyer le cache manuellement, vous pouvez le faire en appelant Cache : :clear(). Cela nettoiera **toutes** les données mises en cache, à l'exception des fichiers de vues mis en cache. Si vous avez besoin de nettoyer les fichiers de vues, utilisez `clearCache()`.

FormHelper

class `FormHelper` (*View $view*, *array $settings = array()*)

Le Helper Form prend en charge la plupart des opérations lourdes en création du formulaire. Le Helper Form se concentre sur la possibilité de créé des formulaires rapidement, d'une manière qui permettra de rationaliser la validation, la re-population et la mise en page (layout). Le Helper Form est aussi flexible - Il va faire à peu près tout pour vous en utilisant les conventions, ou vous pouvez utiliser des méthodes spécifiques pour ne prendre uniquement que ce dont vous avez besoin.

Création de Formulaire

La première méthode dont vous aurez besoin d'utiliser pour prendre pleinement avantage du Helper Form (Helper Formulaire) est `create()`. Cette méthode affichera un tag d'ouverture de formulaire.

`FormHelper::`**`create`** (*string $model = null*, *array $options = array()*)

Tous les paramètres sont optionnels. Si create() est appelée sans paramètres, CakePHP supposera que vous voulez créer un formulaire en rapport avec le controller courant, ou l'URL actuelle. La méthode par défaut pour les formulaires est POST. L'élément du formulaire est également renvoyé avec un DOM ID. Cet identifiant est créé à partir du nom du model, et du nom du controller en notation CamelCase (les majuscules délimitent les mots). Si j'appelle `create()` dans une vue de UsersController, j'obtiendrai ce genre de rendu dans ma vue :

```
<form id="UserAddForm" method="post" action="/users/add">
```

Note : Vous pouvez aussi passer `false` pour `$model`. Ceci placera vos donnée de formulaire dans le tableau : `$this->request->data` (au lieu du sous tableau :`$this->request->data['Model']`). Cela peut être pratique pour des formulaires courts qui ne représenteraient rien dans votre base de données.

La méthode `create()` nous permet également de personnaliser plusieurs paramètres. Premièrement, vous pouvez spécifier un nom de model. Ce faisant, vous modifiez le *contexte* de ce formulaire. Tous

les champs seront supposés dépendre de ce model (sauf si spécifié), et tous les models devront être liés à lui. Si vous ne spécifiez pas de model, CakePHP supposera que vous utilisez le model par défaut pour le controller courant. :

```
// si vous êtes sur /recipes/add
echo $this->Form->create('Recipe');
```

Affichera :

```
<form id="RecipeAddForm" method="post" action="/recipes/add">
```

Ce formulaire enverra les données à votre action `add()` de RecipesController (RecettesController) . Cependant, vous pouvez utiliser la même logique pour créer et modifier des formulaires. Le helper Form utilise la propriété `$this->request->data` pour détecter automatiquement s'il faut créer un formulaire d'ajout ou de modification. Si `$this->request->data` contient un tableau nommé après le model du formulaire , et que ce tableau contient une valeur non nulle pour la clé primaire du model, alors le FormHelper créera un formulaire de modification pour cet enregistrement précis. Par exemple, si on va à l'adresse http ://site.com/recipes/edit/5, nous pourrions avoir cela :

```
// Controller/RecipesController.php:
public function edit($id = null) {
    if (empty($this->request->data)) {
        $this->request->data = $this->Recipe->findById($id);
    } else {
        // La logique de sauvegarde se fera ici
    }
}

// View/Recipes/edit.ctp:
// Puisque $this->request->data['Recipe']['id'] = 5, nous aurons un formulaire d'éditi
echo $this->Form->create('Recipe');
```

Affichera :

```
<form id="RecipeEditForm" method="post" action="/recipes/edit/5">
<input type="hidden" name="_method" value="PUT" />
```

Note : Comme c'est un formulaire de modification, un champ caché (hidden) est créé pour réécrire la méthode HTTP par défaut

---

A la création de formulaires pour les models dans des plugins. Nous devrons toujours utiliser la notation *plugin syntax* à la création d'un formulaire. Cela assurera que le formulaire est correctement généré :

```
echo $this->Form->create('ContactManager.Contact');
```

Le tableau `$options` est l'endroit où la plupart des paramètres de configurations sont stockés. Ce tableau spécial peut contenir un certain nombre de paires clé-valeur qui peuvent affecter la manière dont le formulaire sera créé. Modifié dans la version 2.0. L'Url par défaut pour tous les formulaires, est maintenant l'Url incluant passed, named, et les paramètres de requête (querystring). Vous pouvez redéfinir cette valeur par défaut en fournissant `$options['url']` en second paramètre de `$this->Form->create()`.

Options pour create() Il y plusieurs options pour create() :

- `$options['type']` Cette clé est utilisée pour spécifier le type de formulaire à créer. Les valeurs que peuvent prendre cette variable sont 'post', 'get', 'file', 'put' et 'delete'.
  Choisir 'post' ou 'get' changera la méthode de soumission du formulaire en fonction de votre choix. :

```
echo $this->Form->create('User', array('type' => 'get'));
```

  Affichera :

```
<form id="UserAddForm" method="get" action="/users/add">
```

  En spécifiant 'file' cela changera la méthode de soumission à 'post', et ajoutera un enctype "multipart/form-data" dans le tag du formulaire. Vous devez l'utiliser si vous avez des demandes de fichiers dans votre formulaire. L'absence de cet attribut enctype empêchera le fonctionnement de l'envoi de fichiers. :

```
echo $this->Form->create('User', array('type' => 'file'));
```

  Affichera :

```
<form id="UserAddForm" enctype="multipart/form-data" method="post" action="/users/add">
```

  Quand vous utilisez 'put' ou 'delete', votre formulaire aura un fonctionnement équivalent à un formulaire de type 'post', mais quand il sera envoyé, la méthode de requête HTTP sera respectivement réécrite avec 'PUT' ou 'DELETE'. Cela permettra à CakePHP de créer son propre support REST dans les navigateurs web.
- `$options['action']` La clé action vous permet de définir vers quelle action de votre controller pointera le formulaire. Par exemple, si vous voulez que le formulaire appelle l'action login() de votre controller courant, vous créeriez le tableau $options comme ceci

```
echo $this->Form->create('User', array('action' => 'login'));
```

  Affichera :

```
<form id="UserLoginForm" method="post" action="/users/login">
</form>
```

- `$options['url']` Si l'action que vous désirez appeler avec le formulaire n'est pas dans le controller courant, vous pouvez spécifier une URL dans le formulaire en utilisant la clé 'url' de votre tableau $options. L'URL ainsi fournie peut être relative à votre application CakePHP

```
echo $this->Form->create(null, array('url' => '/recipes/add'));
// ou
echo $this->Form->create(null, array(
    'url' => array('controller' => 'recipes', 'action' => 'add')
));
```

  Affichera :

```
<form method="post" action="/recipes/add">
```

  ou pointer vers un domaine extérieur :

```
echo $this->Form->create(null, array(
    'url' => 'http://www.google.com/search',
    'type' => 'get'
));
```

Affichera :

```
<form method="get" action="http://www.google.com/search">
```

Regardez aussi la méthode `HtmlHelper::url()` pour plus d'exemples sur les différents types d'Urls.

- `$options['default']` Si la variable 'default' est définie à false, l'action de soumission du formulaire est changée de telle manière que le bouton submit (de soumission) ne soumet plus le formulaire. Si le formulaire a été créé pour être soumis par AJAX, mettre la variable 'default' à FALSE supprime le comportement par défaut du formulaire, ainsi vous pouvez collecter les données et les soumettre par AJAX à la place.
- `$options['inputDefaults']` Vous pouvez déclarer un jeu d'options par défaut pour `input()` avec la clé `inputDefaults` pour personnaliser vos input par défaut :

```
echo $this->Form->create('User', array(
    'inputDefaults' => array(
     'label' => false,
     'div' => false
    )
 ));
```

Tous les input créés à partir de ce point hériteraient des options déclarées dans inputDefaults. Vous pouvez redéfinir le defaultOptions en déclarant l'option dans l'appel input() :

```
echo $this->Form->input('password'); // Pas de div, Pas de label
echo $this->Form->input('username', array('label' => 'Username')); // a un élément label
```

Fermer le Formulaire

`FormHelper::end`(*$options = null*)

Le FormHelper inclut également une méthode `end()` qui complète le marquage du formulaire. Souvent, `end()` affiche juste la base fermante du formulaire, mais l'utilisation de `end()` permet également au FormHelper d'ajouter les champs cachées dont le component Security `SecurityComponent` à besoin. :

```
<?php echo $this->Form->create(); ?>

<!-- Ici les éléments de Formulaire -->

<?php echo $this->Form->end(); ?>
```

Si une chaîne est fournie comme premier argument à end(), le FormHelper affichera un bouton submit nommé en conséquence en même temps que la balise de fermeture du formulaire. :

```
echo $this->Form->end('Termine');
```

Affichera :

```
<div class="submit">
    <input type="submit" value="Termine" />
</div>
</form>
```

Vous pouvez spécifier des paramètres détaillés en passant un tableau à `end()` :

```
$options = array(
    'label' => 'Update',
    'div' => array(
        'class' => 'glass-pill',
    )
);
echo $this->Form->end($options);
```

Affichera :

```
<div class="glass-pill"><input type="submit" value="Update!" name="Update"></div>
```

Voir l'API (http ://api20.cakephp.org) pour plus de détails.

Note : si vous utilisez le component sécurité `SecurityComponent` dans votre application vous devez toujours terminer vos formulaires avec `end()`.

Création d'éléments de Formulaire

Il y a plusieurs façons pour créer des Forms inputs (entrée de formulaire) Commençons par regarder `input()`. Cette méthode inspecte automatiquement le champ du model qui lui est fourni afin de créer une entrée appropriée pour ce champ. En interne `input()` délègue aux autre méthode du FormHelper.

`FormHelper::input` (*string $fieldName*, *array $options = array()*)

Crée les éléments suivants en donnant un `Model.field` particulier :
–div enveloppante (wrapping div).
–label de l'élément (Label element)
–input de(s) l'élément(s) (Input element(s))
–Erreur de l'élément avec un message si c'est applicable.
Le type d'input créés dépends de la colonne datatype :

Column Type Champ de formulaire résultant

string (char, varchar, etc.) text

boolean, tinyint(1) checkbox

text textarea

text, avec le nom de password, passwd, ou psword password

text, avec le nom de email email

text, avec le nom de tel, telephone, ou phone tel

date day, month, et year selects

datetime, timestamp day, month, year, hour, minute, et meridian selects

time hour, minute, et meridian selects

Le paramètre `$options` vous permet de personnaliser le fonctionnement de `input()`, et contrôle finement ce qui est généré.

Le div entourant aura un nom de classe `required` ajouté à la suite si les règles de validation pour le champ du Model ne spécifient pas `allowEmpty => true`. Une limitation de ce comportement est que le champ du model doit avoir été chargé pendant la requête.

Ou être directement associé au model fourni par `create()`. Introduit dans la version 2.3. Depuis 2.3, l'attribut HTML5 `required` va aussi être ajouté selon les règles de validation du champ. Vous pouvez explicitement définir la clé `required` dans le tableau d'options pour la surcharger pour un champ. Pour échapper la validation attrapée par le navigateur pour l'ensemble du formulaire, vous pouvez définir l'option `'formnovalidate' => true` pour l'input button que vous générez en utilisant `FormHelper::submit()` ou définir `'novalidate' => true` dans les options pour `FormHelper::create()`.

Par exemple, supposons que votre model User contient les champs username (varchar), password (varchar), approved (datetime) et quote (text). Vous pouvez utiliser la méthode input() de l'Helper Formulaire (Formhelper) pour créer une entrée appropriée pour tous les champs du formulaire. :

```
    echo $this->Form->create();

    echo $this->Form->input('username');    //text
    echo $this->Form->input('password');    //password
    echo $this->Form->input('approved');    //day, month, year, hour, minute, meridian
    echo $this->Form->input('quote');       //textarea

    echo $this->Form->end('Add');

Un exemple plus complet montrant quelques options pour le champ de date ::

    echo $this->Form->input('birth_dt', array(
        'label' => 'Date de naissance',
        'dateFormat' => 'DMY',
        'minYear' => date('Y') - 70,
        'maxYear' => date('Y') - 18,
    ));

Outre les options spécifique pour ``input()`` vu ci-dessus, vous pouvez
spécifier n'importe quelle options pour le type d'input et n'importe quel
attribut html (actuellement dans le focus).
Pour plus d'information sur les ``$options`` et ``$htmlAttributes`` voir
:doc:'/core-libraries/helpers/html'.

Supposons un User hasAndBelongsToMany Group. Dans votre controller,
définissez une variable camelCase au pluriel (groupe -> groupes dans cette
exemple, ou ExtraFunkyModele -> extraFunkyModeles) avec les options de
sélections. Dans l'action du controller vous pouvez définir ::

    $this->set('groups', $this->User->Group->find('list'));

Et dans la vue une sélection multiple peut être crée avec ce simple code::

    echo $this->Form->input('Group');

Si vous voulez un champ de sélection utilisant une relation belongsTo
ou hasOne, vous pouvez ajouter ceci dans votre controller Users
(en supposant que l'User belongsTo Group)::

    $this->set('groups', $this->User->Group->find('list'));
```

```
Ensuite, ajouter les lignes suivantes à votre vue de formulaire::

    echo $this->Form->input('group_id');

Si votre nom de model est composé de deux mots ou plus,
ex. "UserGroup", quand vous passez les données en utilisant set()
vous devrez nommer vos données dans un format CamelCase
(les Majuscules séparent les mots) et au pluriel comme ceci ::

    $this->set('userGroups', $this->UserGroup->find('list'));
    // ou bien
    $this->set('reallyInappropriateModelNames', $this->ReallyInappropriateModelName->f

.. note::

    Essayez d'éviter l'utilisation de `FormHelper::input()` pour générer
    les boutons submit. Utilisez plutôt :php:meth:`FormHelper::submit()`.
```

`FormHelper::inputs` (*mixed $fields = null, array $blacklist = null*)

Génère un ensemble d'inputs (entrées) pour `$fields`. Si $fields est null, le model courant sera utilisé.

En plus de l' affichage des champs de controller, `$fields` peut être utilisé pour contrôler legend et fieldset (jeu de champs) rendus avec les clés `fieldset` et `legend`. `$form->inputs(array('legend' => 'Ma légende'));` Générera un jeu de champs input avec une légende personnalisée. Vous pouvez personnaliser des champs input individuels a travers `$fields` comme ceci. :

```
echo $form->inputs(array(
    'name' => array('label' => 'label perso')
));
```

En plus des champs de contrôle (fields control), inputs() permet d'utiliser quelques options supplémentaires.

- `fieldset` Mis à false pour désactiver le jeu de champs (fieldset). Si une chaîne est fournit, elle sera utilisée comme nom de classe (classname) pour l'élément fieldset.
- `legend` Mis à false pour désactiver la légende (legend) pour le jeu de champs input (input set) généré. Ou fournit une chaîne pour personnaliser le texte de la légende (legend).

Conventions de nommage des champs Le Helper Form est assez évolué. Lorsque vous définissez un nom de champ avec les méthodes du Helper Form, celui-ci génère automatiquement une balise input basée sur le nom de model courant, selon le format suivant :

```
<input type="text" id="ModelnameFieldname" name="data[Modelname][fieldname]">
```

Ceci permet d'omettre le nom du model lors de la génération des inputs du model pour lequel le formulaire à été créé. Vous pouvez créez des inputs pour des models associés , ou des models arbitraires en passant dans Modelname.fieldname comme premier paramètre :

```
echo $this->Form->input('Modelname.fieldname');
```

Si vous avez besoin de spécifier de multiples champs en utilisant le même nom de champ, créant ainsi un tableau qui peut être sauver en un coup avec saveAll(), utilisez les conventions suivantes :

```
echo $this->Form->input('Modelname.0.fieldname');
echo $this->Form->input('Modelname.1.fieldname');
```

Affichera :

```
<input type="text" id="Modelname0Fieldname" name="data[Modelname][0][fieldname]">
<input type="text" id="Modelname1Fieldname" name="data[Modelname][1][fieldname]">
```

Le Helper Form utilise plusieurs suffixes de champ en interne pour la création de champ input datetime. Si vous utilisez des champs nommés `year`, `month`, `day`, `hour`, `minute`, or `meridian` et rencontrez des problèmes pour obtenir un input correct, vous pouvez définir le nom `name` de l'attribut pour remplacer le behavior par défaut :

```
echo $this->Form->input('Model.year', array(
    'type' => 'text',
    'name' => 'data[Model][year]'
));
```

Options `FormHelper::input()` supporte un nombre important d'options. En plus de ses propres options `input()` accepte des options pour les champs input générés, comme les attributs html. Ce qui suit va couvrir les options spécifiques de `FormHelper::input()`.

– `$options['type']` Vous pouvez forcer le type d'un input, remplaçant l'introspection du model, en spécifiant un type. En plus des types de champs vus dans *Création d'éléments de Formulaire*, vous pouvez aussi créez des 'fichiers', 'password' et divers types supportés par HTML5 :

```
echo $this->Form->input('field', array('type' => 'file'));
echo $this->Form->input('email', array('type' => 'email'));
```

Affichera :

```
<div class="input file">
    <label for="UserField">Field</label>
    <input type="file" name="data[User][field]" value="" id="UserField" />
</div>
<div class="input email">
    <label for="UserEmail">Email</label>
    <input type="email" name="data[User][email]" value="" id="UserEmail" />
</div>
```

– `$options['div']` Utilisez cette option pour définir les attributs de la div contentant l'input. En utilisant une valeur chaîne configurera le nom de classe de la div. Un tableau clés/valeurs paramétrera les attributs de la div. Alternativement, vous pouvez définir cet clé à false pour désactiver le rendu de la div. Définir le nom de classe :

```
echo $this->Form->input('User.name', array(
    'div' => 'class_name'
));
```

Affichera :

```
<div class="class_name">
    <label for="UserName">Name</label>
    <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Paramétrage de plusieurs attibuts :

```
echo $this->Form->input('User.name', array(
    'div' => array(
      'id' => 'mainDiv',
     'title' => 'Div Title',
       'style' => 'display:block'
 )
));
```

Affichera :

```
<div class="input text" id="mainDiv" title="Div Title" style="display:block">
    <label for="UserName">Name</label>
    <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Désactiver le rendu de la div

```
echo $this->Form->input('User.name', array('div' => false)); ?>
```

Affichera :

```
<label for="UserName">Name</label>
<input name="data[User][name]" type="text" value="" id="UserName" />
```

– `$options['label']` Définissez cette clé à la chaîne que vous voudriez afficher dans le label qui accompagne le input :

```
echo $this->Form->input('User.name', array(
    'label' => 'Alias de l'user'
));
```

Affichera :

```
<div class="input">
    <label for="UserName">Alias de l'user</label>
    <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Alternativement, définissez cette clé à false pour désactiver le rendu du label :

```
echo $this->Form->input('User.name', array('label' => false));
```

Affichera :

```
<div class="input">
    <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Définissez ceci dans un tableau pour fournir des options supplémentaires pour l'élément `label`. Si vous faites cela, vous pouvez utiliser une clé `text` dans le tableau pour personnaliser le texte du label :

```
echo $this->Form->input('User.name', array(
    'label' => array(
        'class' => 'bidule',
        'text' => 'le traducteur est fou hihaaarrrr!!!'
    )
));
```

Affichera :

```
<div class="input">
    <label for="UserName" class="bidule">le traducteur est fou hihaaarrrr!!!</label>
    <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

– `$options['error']` En utilisant cette clé vous permettra de transformer les messages de model par défaut et de les utiliser, par exemple, pour définir des messages i18n. (cf internationalisation). comporte un nombre de sous-options qui contrôles l'enveloppe de l'élément (wrapping) . Le nom de classe de l'élément enveloppé, ainsi que les messages d'erreurs qui contiennent du HTML devront être échappés. Pour désactiver le rendu des messages d'erreurs définissez la clé error à false :

```
$this->Form->input('Model.field', array('error' => false));
```

Pour modifier le type d'enveloppe de l'élément et sa classe, utilisez le format suivant :

```
$this->Form->input('Model.field', array(
    'error' => array('attributes' => array('wrap' => 'span', 'class' => 'bzzz'))
));
```

Pour éviter que le code HTML soit automatiquement échappé dans le rendu du message d'erreur, définissez la sous-option escape à false :

```
$this->Form->input('Model.field', array(
    'attributes' => array('escape' => false)
));
```

Pour surcharger les messages d'erreurs du model utilisez un tableau avec les clés respectant les règles de validation :

```
$this->Form->input('Model.field', array(
    'error' => array('tooShort' => __('Ceci n'est pas assez long'))
));
```

Comme vu ci-dessus vous pouvez définir les messages d'erreurs pour chacune des règles de validation de vos models. Vous pouvez de plus fournir des messages i18n pour vos formulaires.

Introduit dans la version 2.3 : Support pour l'option `errorMessage` a été ajouté dans 2.3

– `$options['before']`, `$options['between']`, `$options['separator']`, et `$options['after']`
Utilisez ces clés si vous avez besoin d'injecter quelques balises à la sortie de la méthode input(). :

```
echo $this->Form->input('field', array(
    'before' => '--avant--',
    'after' => '--après--',
    'between' => '--entre---'
));
```

Affichera :

```
<div class="input">
--avant--
<label for="UserField">Field</label>
--entre---
<input name="data[User][field]" type="text" value="" id="UserField" />
--après--
</div>
```

Pour les input de type radio l'attribut 'separator' peut être utilisé pour injecter des balise pour séparer input/label. :

```
  echo $this->Form->input('field', array(
    'before' => '--avant--',
    'after' => '--après--',
    'between' => '--entre---',
    'separator' => '--séparateur--',
    'options' => array('1', '2')
));
```

Affichera :

```
<div class="input">
--avant--
<input name="data[User][field]" type="radio" value="1" id="UserField1" />
<label for="UserField1">1</label>
--séparateur--
<input name="data[User][field]" type="radio" value="2" id="UserField2" />
<label for="UserField2">2</label>
--entre---
--après--
</div>
```

Pour un élément de type `date` et `datetime` l'attribut 'separator' peut être utilisé pour modifier la chaîne entre les select. Par défaut '-'.

- `$options['format']` L'ordre du code HTML généré par FormHelper est contrôlable comme vous le souhaitez. l'option 'format' supporte un tableau de chaîne décrivant le model de page que vous voudriez que l'élément suive. Les clés de tableau supportées sont :

```
array('before', 'input', 'between', 'label', 'after','error')
```

- `$options['inputDefaults']` S'il vous semble répéter la même option dans de multiples appels input(), vous pouvez utiliser `inputDefaults` pour garder un code propre. :

```
echo $this->Form->create('User', array(
    'inputDefaults' => array(
        'label' => false,
        'div' => false
    )
));
```

Tous les inputs créés a partir de ce point hériterons des valeurs déclarées dans inputDefaults. Vous pouvez redéfinir defaultOptions en déclarant l'option dans l'appel de l'input() :

```
// Pas de div, ni label
echo $this->Form->input('password');

// a un élément label
echo $this->Form->input('username', array('label' => 'Username'));
```

Si vous avez besoin de changer plus tard les valeurs par défaut, vous pourrez utiliser `FormHelper::inputDefaults()`.

Générer des types de inputs spécifiques

En plus de la méthode générique `input()`, le `FormHelper` à des méthodes spécifiques pour générer différents types d'inputs. Ceci peut être utilisé pour générer juste un extrait de code input, et combiné avec d'autres méthodes comme `label()` et `error()` pour générer des layouts (mise en page) complètements personnalisées.

Options Communes Beaucoup des différentes méthodes d'input supportent un jeu d'options communes. Toutes ses options sont aussi supportés par `input()`. Pour réduire les répétitions les options communes partagées par toutes les méthodes input sont :

- `$options['class']` Vous pouvez définir le nom de classe pour un input :

```
echo $this->Form->input('title', array('class' => 'class-perso'));
```

- `$options['id']` Définir cette clé pour forcer la valeur du DOM id pour cet input.
- `$options['default']` Utilisé pour définir une valeur par défaut au champ input. La valeur est utilisée si les données passées au formulaire ne contiennent pas de valeur pour le champ (ou si aucune donnée n'est transmise)
  Exemple d'utilisation :

```
echo $this->Form->input('ingredient', array('default' => 'Sucre'));
```

Exemple avec un champ sélectionné (Taille "Moyen" sera sélectionné par défaut) :

```
$sizes = array('s' => 'Small', 'm' => 'Medium', 'l' => 'Large');
echo $this->Form->input('size', array('options' => $sizes, 'default' => 'm'));
```

Note : Vous ne pouvez pas utiliser `default` pour sélectionner une chekbox - vous devez plutôt définir cette valeur dans `$this->request->data` dans votre controller, ou définir l'option `checked` de input à true.
La valeur par défaut des champs Date et datetime peut être définie en utilisant la clé 'selected'.
Attention à l'utilisation de false pour assigner une valeur par défaut. Une valeur false est utilisé pour désactiver/exclure les options d'un champ, ainsi `'default' => false` ne définirait aucune valeur. A la place, utilisez `'default' => 0`.

En plus des options ci-dessus, vous pouvez mixer n'importe quel attribut html que vous souhaitez utiliser. Chacun des nom d'options non-special sera traité comme un attribut HTML, et appliqué a l'élément HTML input généré. NdT. celui qui capte cette phrase gagne un giroTermoOnduleur a double convection.

Les options pour select, checkbox et inputs radio

– `$options['selected']` Utilisé en combinaison avec un input de type select (ex. Pour les types select, date, heure, datetime) . Définissez 'selected' pour définir l'élément que vous souhaiteriez définir par défaut au rendu de l'input :

```
echo $this->Form->input('heure_fermeture', array(
    'type' => 'time',
    'selected' => '13:30:00'
));
```

Note : La clé selected pour les inputs de type date et datetime peuvent aussi être des timestamps UNIX.

– `$options['empty']` Est défini à true, pour forcer l'input à rester vide.
Quand passé à une list select (liste de selection), ceci créera une option vide avec une valeur vide dans la liste déroulante. Si vous voulez une valeur vide avec un texte affiché ou juste une option vide, passer une chaîne pour vider :

```
  echo $this->Form->input('field', array(
      'options' => array(1, 2, 3, 4, 5),
      'empty' => '(choisissez)'
  ));

Sortie:

.. code-block:: html

  <div class="input">
      <label for="UserField">Field</label>
      <select name="data[User][field]" id="UserField">
          <option value="">(choisissez)</option>
          <option value="0">1</option>
          <option value="1">2</option>
          <option value="2">3</option>
          <option value="3">4</option>
          <option value="4">5</option>
      </select>
  </div>
```

Note :

Si vous avez besoin de définir la valeur par défaut d'un champ password à vide, utilisez 'value'=> '' (deux fois simple cote) à la place.

Les Options peuvent aussi fournir une paire de clé-valeur.

– `$options['hiddenField']` Pour certain types d' input (checkboxes, radios) un input caché est créé ainsi la clé dans $this->request->data existera même sans valeur spécifiée :

```
<input type="hidden" name="data[Post][Published]" id="PostPublished_" value="0" />
<input type="checkbox" name="data[Post][Published]" value="1" id="PostPublished" />
```

Ceci peut être désactivé en définissant l'option `$options['hiddenField'] = false`:

```
echo $this->Form->checkbox('published', array('hiddenField' => false));
```

Retournera :

```
<input type="checkbox" name="data[Post][Published]" value="1" id="PostPublished" />
```

Si vous voulez créer de multiples blocs d'entrés regroupés ensemble dans un formulaire, vous devriez utiliser ce paramètre sur tous les inputs excepté le premier. Si le input caché est en place à différents endroits c'est seulement le dernier groupe de valeur d'input qui sera sauvegardé.
Dans cet exemple , seules les couleurs tertiaires seront passées, et les couleurs primaires seront réécrite :

```
<h2>Couleurs Primaires</h2>
<input type="hidden" name="data[Color][Color]" id="Couleurs_" value="0" />
<input type="checkbox" name="data[Color][Color][]" value="5" id="CouleursRouges" />
<label for="CouleursRouges">Rouge</label>
<input type="checkbox" name="data[Color][Color][]" value="5" id="CouleursBleus" />
<label for="CouleursBleus">Bleu</label>
<input type="checkbox" name="data[Color][Color][]" value="5" id="CouleursJaunes" />
<label for="CouleursJaunes">Jaune</label>

<h2>Couleurs Tertiaires</h2>
<input type="hidden" name="data[Color][Color]" id="Couleurs_" value="0" />
<input type="checkbox" name="data[Color][Color][]" value="5" id="CouleursVertes" />
<label for="CouleursVertes">Vert</label>
<input type="checkbox" name="data[Color][Color][]" value="5" id="CouleursPourpres" />
<label for="CouleursPourpres">Pourpre</label>
<input type="checkbox" name="data[Addon][Addon][]" value="5" id="CouleursOranges" />
<label for="CouleursOranges">Orange</label>
```

En désactivant le champ caché `'hiddenField'` dans le second groupe d'input empêchera ce behavior.
Vous pouvez définir une valeur différente pour le champ caché autre que 0 comme 'N' :

```
echo $this->Form->checkbox('published', array(
    'value' => 'Y',
    'hiddenField' => 'N',
));
```

Les options de Datetime

- `$options['timeFormat']`. Utilisé pour spécifier le format des inputs select (menu de sélection) pour un jeu d'input en relation avec le temps. Les valeurs valides sont `12`, `24`, et `null`.
- `$options['dateFormat']` Utilisé pour spécifier le format des inputs select (menu de sélection) pour un jeu d'input en relation avec le temps. Les valeurs valides comprennent n'importe quelle combinaison de 'D', 'M' et 'Y' or `null`. Les input seront placés dans l'ordre définit par l'option dateFormat.
- `$options['minYear']`, `$options['maxYear']` Utilisé en combinaison avec un input date/datetime. Définit les valeurs minimales et/ou maximales de fin montrées dans le champ select years.
- `$options['orderYear']` Utilisé en combinaison avec un input date/datetime. Définit l'ordre dans lequel la valeur de l'année sera délivré. Les valeurs valides sont 'asc', 'desc'. La valeur par défaut est 'desc'.
- `$options['interval']` Cette option spécifie l'écart de minutes entre chaque option dans la select box minute :

```
echo $this->Form->input('Model.time', array(
    'type' => 'time',
    'interval' => 15
));
```

Créera 4 options dans la select box minute. Une toute les 15 minutes.

Éléments de Formulaire-Méthodes spécifiques

`FormHelper::label`(*string $fieldName*, *string $text*, *array $options*)

Crée un élément label. `$fieldName` est utilisé pour générer le Dom id. Si `$text` n'est pas défini, `$fieldName` sera utilisé pour définir le texte du label :

```
echo $this->Form->label('User.name');
echo $this->Form->label('User.name', 'Your username');
```

Affichera :

```
<label for="UserName">Name</label>
<label for="UserName">Your username</label>
```

`$options` peut soit être un tableau d'attributs html, ou une chaîne qui sera utilisée comme nom de classe :

```
echo $this->Form->label('User.name', null, array('id' => 'user-label'));
echo $this->Form->label('User.name', 'Your username', 'highlight');
```

Affichera :

```
<label for="UserName" id="user-label">Name</label>
<label for="UserName" class="highlight">Your username</label>
```

`FormHelper::text`(*string $name*, *array $options*)

Les autres méthodes disponibles dans l'Helper Form permettent la création d'éléments spécifiques de formulaire. La plupart de ces méthodes utilisent également un paramètre spécial $options. Toutefois, dans ce cas, $options est utilisé avant tout pour spécifier les attributs des balises HTML (comme la valeur ou l'id DOM d'un élément du formulaire). :

```
echo $this->Form->text('username', array('class' => 'users'));
```

Affichera :

```
<input name="data[User][username]" type="text" class="users" id="UserUsername" />
```

`FormHelper::password`(*string $fieldName*, *array $options*)

Création d'un champ password. :

```
echo $this->Form->password('password');
```

Affichera :

```
<input name="data[User][password]" value="" id="UserPassword" type="password">
```

`FormHelper::hidden` (*string $fieldName*, *array $options*)

Créera un form input caché. Exemple :

```
echo $this->Form->hidden('id');
```

Affichera :

```
<input name="data[User][id]" value="10" id="UserId" type="hidden">
```

Modifié dans la version 2.0 : Les champs cachés n'enlève plus la classe attribute. Cela signifie que si il y a des erreurs de validation sur les champs cachés, le nom de classe error-field sera appliqué.

`FormHelper::textarea` (*string $fieldName*, *array $options*)

Crée un champ input textarea (zone de texte). :

```
echo $this->Form->textarea('notes');
```

Affichera :

```
<textarea name="data[User][notes]" id="UserNotes"></textarea>
```

Note : Le input `textarea` permet pour `$options` l'attribut `'escape'` lequel détermine si oui ou non le contenu du textarea pourrait être échappé. Par défaut à `true`.

```
echo $this->Form->textarea('notes', array('escape' => false);
// OU....
echo $this->Form->input('notes', array('type' => 'textarea', 'escape' => false);
```

Options

En plus de *Options Communes*, textarea() supporte quelques options spécifiques :

–`$options['rows']`, `$options['cols']` Ces deux clés spécifient le nombre de lignes et de colonnes :

```
echo $this->Form->textarea('textarea', array('rows' => '5', 'cols' => '5'));
```

Affichera :

```
<textarea name="data[Form][textarea]" cols="5" rows="5" id="FormTextarea">
</textarea>
```

`FormHelper::checkbox` (*string $fieldName*, *array $options*)

Crée un élément de formulaire checkbox. Cette méthode génère également un input de formulaire caché pour forcer la soumission de données pour le champ spécifié. :

```
echo $this->Form->checkbox('done');
```

Affichera :

```
<input type="hidden" name="data[User][done]" value="0" id="UserDone_" />
<input type="checkbox" name="data[User][done]" value="1" id="UserDone" />
```

Il est possible de modifier la valeur du checkbox en utilisant le tableau $options :

```
echo $this->Form->checkbox('done', array('value' => 555));
```

Affichera :

```
<input type="hidden" name="data[User][done]" value="0" id="UserDone_" />
<input type="checkbox" name="data[User][done]" value="555" id="UserDone" />
```

Si vous ne voulez pas que le Helper Form génère un input caché :

```
echo $this->Form->checkbox('done', array('hiddenField' => false));
```

Affichera :

```
<input type="checkbox" name="data[User][done]" value="1" id="UserDone" />
```

`FormHelper::radio`(*string $fieldName*, *array $options*, *array $attributes*)

Crée un jeu d'inputs radios.

Options

–`$attributes['value']` pour définir quelle valeur sera sélectionnée par défaut.

–`$attributes['separator']` pour spécifier du HTML entre les boutons (ex <br />).

–`$attributes['between']` spécifie quelques contenus à insérer entre la légende et le premier argument.

–`$attributes['disabled']` définit a `true` ou `'disabled'` désactivera tous les boutons radios générés.

–`$attributes['legend']` Les éléments Radio sont enveloppés avec un label et un fieldset (jeu de champs) par défaut. Définir `$attributes['legend']` à false pour les retirer. :

```
$options = array('H' => 'Homme', 'F' => 'Femme');
$attributes = array('legend' => false);
echo $this->Form->radio('genre', $options, $attributes);
```

Affichera :

```
<input name="data[User][genre]" id="UserGenre_" value="" type="hidden">
<input name="data[User][genre]" id="UserGenreH" value="H" type="radio">
<label for="UserGenreH">Homme</label>
<input name="data[User][genre]" id="UserGenreF" value="F" type="radio">
<label for="UserGenreF">Femme</label>
```

Si pour quelque raisons vous ne voulez pas du input caché, définissez `$attributes['value']` à une valeur sélectionnée ou le booléen false Modifié dans la version 2.1 : L'option d'attribut `$attributes['disabled']` a été ajoutée dans Cake 2.1.

`FormHelper::select`(*string $fieldName*, *array $options*, *array $attributes*)

Crée un menu de sélection, rempli des éléments compris dans `$options`, avec l'option spécifiée par `$attributes['value']` sera montré comme sélectionné par défaut. Définir à false la clé 'empty' dans la variable `$attributes` pour empêcher l'option empty par défaut :

```
$options = array('H' => 'Homme', 'F' => 'Femme');
echo $this->Form->select('genre', $options)
```

Affichera :

```
<select name="data[User][genre]" id="UserGenre">
<option value=""></option>
<option value="H">Homme</option>
<option value="F">Femme</option>
</select>
```

L'input de type `select` permet un attribut `$option` spécial appelée `'escape'` qui accepte un booléen et détermine si il faut que l'entité HTML encode le contenu des options sélectionnées. Par défaut à true :

```
$options = array('H' => 'Homme', 'F' => 'Femme');
echo $this->Form->select('genre', $options, array('escape' => false));
```

–`$attributes['options']` Cette clé vous permets de spécifier manuellement des options pour un input select (menu de sélection), ou pour un groupe radio. A moins que le 'type' soit spécifié à 'radio', le Helper Form supposera que la cible est un input select (menu de sélection)

```
echo $this->Form->select('field', array(1,2,3,4,5));
```

Affichera :

```
<select name="data[User][field]" id="UserField">
    <option value="0">1</option>
    <option value="1">2</option>
    <option value="2">3</option>
    <option value="3">4</option>
    <option value="4">5</option>
</select>
```

Les options peuvent aussi être fournies comme des paires clé-valeur :

```
echo $this->Form->select('field', $options, array(
    'Value 1' => 'Label 1',
    'Value 2' => 'Label 2',
    'Value 3' => 'Label 3'
));
```

Affichera :

```
<select name="data[User][field]" id="UserField">
    <option value="Value 1">Label 1</option>
    <option value="Value 2">Label 2</option>
    <option value="Value 3">Label 3</option>
</select>
```

Si vous souhaitez générer un select avec des groupes optionnels, passez les données dans un format hiérarchique. Ceci fonctionnera avec les checkboxes multiples et les boutons radios également, mais au lieu des groupes optionnels enveloppez les éléments dans des fieldsets :

```
$options = array(
   'Group 1' => array(
      'Value 1' => 'Label 1',
      'Value 2' => 'Label 2'
   ),
   'Group 2' => array(
      'Value 3' => 'Label 3'
```

```
    )
);
echo $this->Form->select('field', $options);
```

Affichera :

```
<select name="data[User][field]" id="UserField">
    <optgroup label="Group 1">
        <option value="Value 1">Label 1</option>
        <option value="Value 2">Label 2</option>
    </optgroup>
    <optgroup label="Group 2">
        <option value="Value 3">Label 3</option>
    </optgroup>
</select>
```

–`$attributes['multiple']` Si 'multiple' a été défini à true pour un input select, celui ci autorisera les sélections multiples :

```
echo $this->Form->select('Model.field', $options, array('multiple' => true));
```

Vous pouvez également définir 'checkbox' à 'multiple' pour afficher une liste de check boxes reliés :

```
$options =  array(
    'Value 1' => 'Label 1',
    'Value 2' => 'Label 2'
);
echo $this->Form->select('Model.field', $options, array(
    'multiple' => 'checkbox'
));
```

Affichera :

```
<div class="input select">
   <label for="ModelField">Field</label>
   <input name="data[Model][field]" value="" id="ModelField" type="hidden">
   <div class="checkbox">
      <input name="data[Model][field][]" value="Value 1" id="ModelField1" type="check
      <label for="ModelField1">Label 1</label>
   </div>
   <div class="checkbox">
      <input name="data[Model][field][]" value="Value 2" id="ModelField2" type="check
      <label for="ModelField2">Label 2</label>
   </div>
</div>
```

–`$attributes['disabled']` Lors de la création de checkboxes, cette option peut être défini pour désactiver tout ou quelques checkboxes. Pour désactiver toutes les checkboxes, définissez disabled à `true` :

```
$options = array(
    'Value 1' => 'Label 1',
    'Value 2' => 'Label 2'
);
echo $this->Form->select('Model.field', $options, array(
    'multiple' => 'checkbox',
```

```
        'disabled' => array('Value 1')
    ));
```

Output :

```
<div class="input select">
   <label for="ModelField">Field</label>
   <input name="data[Model][field]" value="" id="ModelField" type="hidden">
   <div class="checkbox">
      <input name="data[Model][field][]" disabled="disabled" value="Value 1" id="Mode
      <label for="ModelField1">Label 1</label>
   </div>
   <div class="checkbox">
      <input name="data[Model][field][]" value="Value 2" id="ModelField2" type="check
      <label for="ModelField2">Label 2</label>
   </div>
</div>
```

Modifié dans la version 2.3 : Le support pour les tableaux dans `$attributes['disabled']` a été ajoutée dans 2.3.

`FormHelper::file`(*string $fieldName*, *array $options*)

Pour ajouter un champ upload à un formulaire, vous devez vous assurer que le enctype du formulaire est définit a "multipart/form-data", donc commençons avec une fonction create comme ci-dessous :

```
echo $this->Form->create('Document', array('enctype' => 'multipart/form-data'));
// OU
echo $this->Form->create('Document', array('type' => 'file'));
```

Ensuite ajoutons l'une ou l'autre des deux lignes dans le fichier de vue de votre formulaire :

```
echo $this->Form->input('Document.submittedfile', array(
    'between' => '<br />',
    'type' => 'file'
));

// OU

echo $this->Form->file('Document.submittedfile');
```

En raisons des limitations du code HTML lui même, il n'est pas possible de placer des valeurs par défauts dans les champs inputs de type 'file'. A chacune des fois ou le formulaire sera affiché, la valeur sera vide.

Lors de la soumission, le champ file fournit un tableau étendu de données au script recevant les données de formulaire.

Pour l'exemple ci-dessus, les valeurs dans le tableau de données soumis devraient être organisées comme à la suite, si CakePHP à été installé sur un server Windows .'tmp_name' aura un chemin différent dans un environnement Unix :

```
$this->request->data['Document']['submittedfile'] = array(
    'name' => conference_schedule.pdf,
    'type' => application/pdf,
    'tmp_name' => C:/WINDOWS/TEMP/php1EE.tmp,
    'error' => 0,
```

```
        'size' => 41737,
    );
```

Ce tableau est généré par PHP lui-même, pour plus de détails sur la façon dont PHP gère les données passées a travers les champs `files`. lire la section file uploads du manuel de PHP (http ://php.net/features.file-upload).

Validation des Uploads Ci dessous l'exemple d'une méthode de validation définit dans votre model pour valider si un fichier à été uploader avec succès :

```
public function isUploadedFile($params) {
    $val = array_shift($params);
    if ((isset($val['error']) && $val['error'] == 0) ||
        (!empty( $val['tmp_name']) && $val['tmp_name'] != 'none')
    ) {
        return is_uploaded_file($val['tmp_name']);
    }
    return false;
}
```

Crée un input file :

```
echo $this->Form->create('User', array('type' => 'file'));
echo $this->Form->file('avatar');
```

Affichera :

```
<form enctype="multipart/form-data" method="post" action="/users/add">
<input name="data[User][avatar]" value="" id="UserAvatar" type="file">
```

Note : Quand vous utilisez `$this->Form->file()`, rappelez-vous de définir le type d'encodage , en définissant l'option de type à 'file' dans `$this->Form->create()`.

Création des boutons et des éléments submits

`FormHelper::`**`submit`**(*string $caption*, *array $options*)

Crée un bouton submit avec la légende `$caption`. Si la `$caption` fournie est l'URL d'une image (il contient un caractère '.'), le bouton submit sera rendu comme une image.

Il est encapsulé entre des `div` par défaut ; vous pouvez empêcher cela en déclarant `$options['div'] = false`:

```
echo $this->Form->submit();
```

Affichera :

```
<div class="submit"><input value="Submit" type="submit"></div>
```

Vous pouvez aussi passer une url relative ou absolue vers une image pour le paramètre caption au lieu d'un caption text :

```
echo $this->Form->submit('ok.png');
```

Affichera :

```
<div class="submit"><input type="image" src="/img/ok.png"></div>
```

`FormHelper::button`(*string $title*, *array $options = array()*)

Crée un boutton HTML avec le titre spécifié et un type par défaut "button". Définir `$options['type']` affichera l'un des trois types de boutons possibles :

1.submit : Comme celui de la méthode `$this->Form->submit`- (par défaut).

2.reset : Crée un bouton reset.

3.button : Crée un bouton standard.

```
echo $this->Form->button('Un bouton');
echo $this->Form->button('Un autre Bouton', array('type' => 'button'));
echo $this->Form->button('Initialise le Formulaire', array('type' => 'reset'));
echo $this->Form->button('Soumettre le Formulaire', array('type' => 'submit'));
```

Affichera :

```
<button type="submit">Un bouton</button>
<button type="button">Un autre Bouton</button>
<button type="reset">Initialise le Formulaire</button>
<button type="submit">Soumettre le Formulaire</button>
```

Le input de type `button` supporte l'option `escape` qui accepte un booléen et détermine si oui ou non l'entité HTML encode le $title du bouton. Par défaut à false :

```
echo $this->Form->button('Submit Form', array('type' => 'submit', 'escape' => true));
```

`FormHelper::postButton`(*string $title*, *mixed $url*, *array $options = array ()*)

Crée un tag''<button>'' avec un `<form>` l'entourant qui soumets à travers POST.

Cette méthode créé un élément `<form>`. Donc n'utilisez pas pas cette méthode dans un formulaire ouvert. Utilisez plutôt `FormHelper::submit()` ou `:php:meth:'FormHelper::button()` pour créér des boutons a l'intérieur de formulaires ouvert.

`FormHelper::postLink`(*string $title*, *mixed $url = null*, *array $options = array ()*, *string $confirmMessage = false*)

Crée un lien HTML, mais accède à l'Url en utilisant la méthode POST. Requiert que javascript soit autorisé dans votre navigateur.

Cette méthode créée un élément `<form>`. Donc n'utilisez pas cette méthode dans un formulaire existant. En remplacement vous devriez ajouter un bouton submit en utilisant `FormHelper::submit()`. Modifié dans la version 2.3. L'option `method` a été ajoutée.

Crée des inputs de date et d'heure (date and time inputs)

`FormHelper::dateTime`(*$fieldName*, *$dateFormat = 'DMY'*, *$timeFormat = '12'*, *$attributes = array()*)

Crée un jeu d'inputs pour la date et l'heure. Les valeurs valides pour $dateformat sont 'DMY', 'MDY', 'YMD' ou 'NONE. Les valeurs valides pour $timeFormat sont '12', '24', et null.

Vous pouvez spécifier de ne pas afficher les valeurs vides en paramétrant "array('empty' => false)" dans les paramètres des attributs. il pré-sélectionnera également les champs a la date et heure courante.

`FormHelper::year`(*string $fieldName*, *int $minYear*, *int $maxYear*, *array $attributes*)

Crée un élément select'(menu de sélection) rempli avec les années depuis `$minYear` jusqu'à `$maxYear`. Les attributs HTML devrons être fournis dans $attributes. Si `$attributes['empty']` est false, le select n'inclura pas d'option empty :

```
echo $this->Form->annee('purchased', 2000, date('Y'));
```

Affichera :

```
<select name="data[User][purchased][annee]" id="UserPurchasedYear">
<option value=""></option>
<option value="2009">2009</option>
<option value="2008">2008</option>
<option value="2007">2007</option>
<option value="2006">2006</option>
<option value="2005">2005</option>
<option value="2004">2004</option>
<option value="2003">2003</option>

<option value="2002">2002</option>
<option value="2001">2001</option>
<option value="2000">2000</option>
</select>
```

`FormHelper::month`(*string $fieldName*, *array $attributes*)

Crée un élément select (menu de sélection) avec le nom des mois :

```
echo $this->Form->month('mob');
```

Affichera :

```
<select name="data[User][mob][month]" id="UserMobMonth">
<option value=""></option>
<option value="01">January</option>
<option value="02">February</option>
<option value="03">March</option>
<option value="04">April</option>
<option value="05">May</option>
<option value="06">June</option>
<option value="07">July</option>
<option value="08">August</option>
<option value="09">September</option>
<option value="10">October</option>
<option value="11">November</option>
<option value="12">December</option>
</select>
```

Vous pouvez passer votre propre tableau des mois à utiliser en paramétrant l'attribut 'monthNames', ou avoir les mois affichés comme des nombres en passant false. (Note : les mois par défaut sont internationalisés et peuvent être traduits en utilisant la localisation) :

```
echo $this->Form->month('mob', null, array('monthNames' => false));
```

`FormHelper::day`(*string $fieldName*, *array $attributes*)

Crée un élément select (menu de sélection) rempli avec les jours (numériques) du mois.

Pour créer une option empty avec l'affichage d'un texte de votre choix (ex. la première option est 'Jour'), vous pouvez fournir le texte comme paramètre final comme ceci :

```
echo $this->Form->day('created');
```

Affichera :

```
<select name="data[User][created][day]" id="UserCreatedDay">
<option value=""></option>
<option value="01">1</option>
<option value="02">2</option>
<option value="03">3</option>
...
<option value="31">31</option>
</select>
```

`FormHelper::hour`(*string $fieldName*, *boolean $format24Hours*, *array $attributes*)

Crée un élément select (menu de sélection) rempli avec les heures de la journée.

`FormHelper::minute`(*string $fieldName*, *array $attributes*)

Crée un élément select (menu de sélection) rempli avec les minutes d'une heure.

`FormHelper::meridian`(*string $fieldName*, *array $attributes*)

Crée un élément select (menu de sélection) rempli avec 'am' et 'pm'.

Afficher et vérifier les erreurs

`FormHelper::error`(*string $fieldName*, *mixed $text*, *array $options*)

Affiche un message d'erreur de validation, spécifiée par $texte, pour le champ donné, dans le cas où une erreur de validation a eu lieu.

Options :

–'escape' booléen si il faut ou non que le html échappe le contenu de l'erreur.

–'wrap' valeur mixte définissant s'il faut ou pas que le message d'erreur soit envelopper d'une div. Si c'est une chaîne , sera utilisé comme le tag HTML à utiliser.

–'class' string Le nom de classe du message d'erreur.

`FormHelper::isFieldError`(*string $fieldName*)

Retourne true si le champ $fieldName fourni a une erreur de validation en cours :

```
if ($this->Form->isFieldError('genre')) {
    echo $this->Form->error('genre');
}
```

Note : En utilisant `FormHelper::input()`, les erreurs sont retournées par défaut.

`FormHelper::tagIsInvalid()`
Retourne false si le champ fourni décrit par l'entité courante ne contient pas d'erreur. Sinon retourne le message de validation.

Configuration par défaut pour tous les champs

Introduit dans la version 2.2. Vous pouvez déclarer un ensemble d'options par défaut pour `input()` en utilisant `FormHelper::inputDefaults()`. Changer les options par défaut vous permet de consolider les options répétées dans un appel à une unique méthode :

```
$this->Form->inputDefaults(array(
        'label' => false,
        'div' => false,
        'class' => 'fancy'
    )
);
```

Tous les champs créés à partir ce point de retour vont hériter des options déclarées dans inputDefaults. Vous pouvez surcharger les options par défaut en déclarant l'option dans l'appel input() :

```
echo $this->Form->input('password'); // Pas de div, pas de label avec la classe 'fancy'
echo $this->Form->input('username', array('label' => 'Username')); // a un élément label a
```

Travailler avec le Component Sécurity

`SecurityComponent` offre plusieurs fonctionnalités qui rendent vos formulaires plus sûres et plus sécurisés. En incluant simplement le component sécurité `SecurityComponent` dans votre controller, vous bénéficierez automatiquement de CSRF (Cross-site request forgery) et des fonctionnalités pour éviter la falsification.

Quand vous utilisez le SecurityComponent (component de sécurité), vous devez toujours fermer vos formulaires en utilisant `FormHelper::end()`. Ceci assurera que les inputs jeton spéciaux `_Token` seront générés.

`FormHelper::unlockField`(*$name*)
Déverrouille un champ en le rendant exempt du hachage (hashing) du `SecurityComponent`. Ceci permet également au champ d'être manipulé par Javascript. Le paramètre `$name` devra être le nom d'entité de l'input :

```
    $this->Form->unlockField('User.id');
```

`FormHelper::secure`(*array $fields = array()*)
Génère un champ caché avec hachage sur le champ utilisé dans le formulaire.

Mises à jour 2.0

$selected parameter removed

Le paramètre `$selected``a été retiré de plusieurs méthodes du Helper Form (FormHelper). Toutes les méthodes supportent désormais un clé ``$attributes['value']` qui devra être utilisée en remplacement de `$selected`. Ce changement simplifie les méthodes du Helper Form, en réduisant le nombre d'arguments, et réduit les duplications que `$selected` créé. Les méthodes sont :
– FormHelper : :select()
– FormHelper : :dateTime()
– FormHelper : :year()
– FormHelper : :month()
– FormHelper : :day()
– FormHelper : :hour()
– FormHelper : :minute()
– FormHelper : :meridian()

L'url par défaut des formulaires est l'action courante

L'url par défaut pour tous les formulaires, est désormais l'url courante incluant passed, named, et les paramètres de la requête (querystring parameters). Vous pouvez redéfinir cette valeur par défaut en fournissant `$options['url']` dans le second paramètre de `$this->Form->create()`.

FormHelper : :hidden()

Les champs cachés n'enlève plus les attributs de classe. Cela signifie que si il y a des erreurs de validation sur les champs cachés le nom de classe error-field sera appliqué.

HTMLHelper

class `HtmlHelper` (*View $view*, *array $settings = array()*)

Le rôle du Helper Html dans CakePHP est de fabriquer les options du HTML plus facilement, plus rapidement. L'utilisation de cet Helper permettra à votre application d'être plus légère bien ancrée et plus flexible de l'endroit ou il est placé en relation avec la racine de votre domaine.

De nombreuses méthodes du Helper Html contiennent un paramètre `$htmlAttributes`, qui vous permet d'insérer un attribut supplémentaire sur vos tags. Voici quelques exemples sur la façon d'utiliser les paramètres $htmlAttributes :

```
Attributs souhaités: <tag class="someClass" />
Paramètre du tableau: array('class' => 'someClass')

Attributs souhaités: <tag name="foo" value="bar" />
Paramètre du tableau:  array('name' => 'foo', 'value' => 'bar')
```

Note : Le Helper html est disponible dans toutes les vues par défaut. Si vous recevez une erreur vous informant qu'il n'est pas disponible, c'est habituellement dû a son nom qui a été oublié de la configuration manuelle de la variable $helpers du controller.

Insertion d'éléments correctement formatés

La tâche la plus importante que le Helper Html accomplit est la création d'un balisage bien formé. N'ayez pas peur de l'utiliser souvent - vous pouvez cacher les vues dans cakePHP pour économiser du temps CPU quand les vues sont rendues et délivrées. Cette section couvrira les méthodes du Helper Html et comment les utiliser.

`HtmlHelper::charset`(*$charset=null*)

Paramètres

- **$charset** (*string*) – Jeu de caractère désiré. S'il est null, la valeur de `App.encoding` sera utilisée.

Utilisé pour créer une balise meta spécifiant le jeu de caractères du document. UTF-8 par défaut.

Exemple d'utilisation :

```
echo $this->Html->charset();
```

Affichera :

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Sinon :

```
echo $this->Html->charset('ISO-8859-1');
```

Affichera :

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
```

`HtmlHelper::css`(*mixed $path*, *array $options = array()*)

Modifié dans la version 2.4.

Paramètres

- **$path** (*mixed*) – Soit une chaîne du fichier css à lier, ou un tableau avec plusieurs fichiers.
- **$options** (*array*) – Un tableau d'options ou *attributs html*.

Créé un ou plusieurs lien(s) vers un feuille de style CSS. Si la clé 'inline' est définie à false dans les paramètres `$options`, les balises de lien seront ajoutés au bloc `css` lequel sera intégré à la balise entête du document.

Vous pouvez utiliser l'option `block` pour contrôler sur lequel des blocs l'élément lié sera ajouté. Par défaut il sera ajouté au bloc `css`.

Si la clé 'rel' dans le tableau `$options` est défini pour 'import', la feuille de style sera importée.

Cette méthode d'inclusion CSS présume que le CSS spécifié se trouve dans le répertoire /app/webroot/css si un chemin ne commence par un '/'. :

```
echo $this->Html->css('forms');
```

Affichera :

```
<link rel="stylesheet" type="text/css" href="/css/forms.css" />
```

Le premier paramètre peut être un tableau pour inclure des fichiers multiples. :

```
echo $this->Html->css(array('forms', 'tables', 'menu'));
```

Affichera :

```
<link rel="stylesheet" type="text/css" href="/css/forms.css" />
<link rel="stylesheet" type="text/css" href="/css/tables.css" />
<link rel="stylesheet" type="text/css" href="/css/menu.css" />
```

Vous pouvez inclure un fichier css depuis un plugin chargé en utilisant *syntaxe de plugin*. Pour inclure `app/Plugin/DebugKit/webroot/css/toolbar.css`, vous pouvez utiliser ce qui suit :

```
echo $this->Html->css('DebugKit.toolbar.css');
```

Si vous voulez inclure un fichier css qui partage un nom avec un plugin chargé vous pouvez faire ce qui suit. Par exemple vous avez un plugin `Blog`, et souhaitez inclure également `app/webroot/css/Blog.common.css` :

```
echo $this->Html->css('Blog.common.css', null, array('plugin' => false));
```

Modifié dans la version 2.1 : L'option `block` a été ajoutée. Le support de *syntaxe de plugin* à été ajouté.

`HtmlHelper::meta`(*string $type*, *string $url = null*, *array $options = array()*)

Paramètres

- **$type** (*string*) – Le type de balise meta désiré.
- **$url** (*mixed*) – L'url de la balise meta, soit une chaîne ou un tableau *routing array*.
- **$options** (*array*) – Un tableau d'attributs *html attributes*.

Cette méthode est pratique pour faire des liens vers des ressources externes comme RSS/Atom feeds et les favicons. Comme avec css(), vous pouvez spécifier si vous voulez l'apparition de la balise en ligne ou l'ajouter au bloc `meta` en définissant la clé 'inline' à false dans les paramètres $attributes, ex. - `array('inline' => false)`.

Si vous définissez l'attribut "type" en utilisant le paramètre $attributes, CakePHP contient certains raccourcis :

type	valeur résultante
html	text/html
rss	application/rss+xml
atom	application/atom+xml
icon	image/x-icon

```
<?php
echo $this->Html->meta(
    'favicon.ico',
    '/favicon.ico',
    array('type' => 'icon')
);
?>
// Output (line breaks added)
<link
    href="http://example.com/favicon.ico"
    title="favicon.ico" type="image/x-icon"
    rel="alternate"
/>
```

```
<?php
echo $this->Html->meta(
    'Comments',
    '/comments/index.rss',
    array('type' => 'rss')
);
?>
// Output (line breaks added)
<link
    href="http://example.com/comments/index.rss"
    title="Comments"
    type="application/rss+xml"
    rel="alternate"
/>
```

Cette méthode peut aussi être utilisée pour ajouter les balise de mots clés et les descriptions. Exemple :

```
.. code-block:: php
```

< ?php echo $this->Html->meta(

'keywords', 'enter any meta keyword here'

) ; ?> // Output <meta name="keywords" content="enter any meta keyword here" />

< ?php echo $this->Html->meta(

'description', 'enter any meta description here'

) ; ?> // Output <meta name="description" content="enter any meta description here" />

Si vous voulez ajouter une balise personnalisée alors le premier paramètre devra être un tableau. Pour ressortir une balise de robots noindex, utilisez le code suivant :

```
echo $this->Html->meta(array('name' => 'robots', 'content' => 'noindex'));
```

Modifié dans la version 2.1 : L'option `block` a été ajoutée.

`HtmlHelper::docType`(*string $type = 'xhtml-strict'*)

Paramètres

– **$type** (*string*) – Le type de doctype fabriqué.

Retourne un balise doctype (X)HTML. Fournissez le doctype en suivant la table suivante :

type	valeur résultante
html4-strict	HTML4 Strict
html4-trans	HTML4 Transitional
html4-frame	HTML4 Frameset
html5	HTML5
xhtml-strict	XHTML1 Strict
xhtml-trans	XHTML1 Transitional
xhtml-frame	XHTML1 Frameset
xhtml11	XHTML1.1

```
echo $this->Html->docType();
// Affichera: <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.

echo $this->Html->docType('html5');
// Affichera: <!DOCTYPE html>

echo $this->Html->docType('html4-trans');
// Affichera: <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://w
```

Modifié dans la version 2.1 : La valeur par défaut de doctype est html5 avec la version 2.1.

`HtmlHelper::style`(*array $data*, *boolean $oneline = true*)

Paramètres

- **$data** (*array*) – Un ensemble de clé => valeurs avec des propriétés CSS.
- **$oneline** (*boolean*) – Le contenu sera sur une seule ligne.

Construit les définitions de style CSS en se basant sur les clés et valeurs du tableau passé à la méthode. Particulièrement pratique si votre fichier CSS est dynamique. :

```
echo $this->Html->style(array(
    'background' => '#633',
    'border-bottom' => '1px solid #000',
    'padding' => '10px'
));
```

Affichera

```
background:#633; border-bottom:1px solid #000; padding:10px;
```

`HtmlHelper::image`(*string $path*, *array $options = array()*)

param string $path Chemin de l'image.

param array $options Un tableau de *attributs html*.

Créé une balise image formatée. Le chemin fournit devra être relatif à /app/webroot/img/. :

```
echo $this->Html->image('cake_logo.png', array('alt' => 'CakePHP'));
```

Affichera :

```
<img src="/img/cake_logo.png" alt="CakePHP" />
```

Pour créer un lien d'image, spécifiez le lien de destination en utilisant l'option `url` dans `$htmlAttributes`. :

```
echo $this->Html->image("recipes/6.jpg", array(
    "alt" => "Brownies",
    'url' => array('controller' => 'recipes', 'action' => 'view', 6)
));
```

Affichera :

```
<a href="/recipes/view/6">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Si vous créez des images dans des mails, ou voulez des chemins absolus pour les images, vous pouvez utiliser l'option `fullBase` :

```
echo $this->Html->image("logo.png", array('fullBase' => true));
```

Affichera :

```
<img src="http://example.com/img/logo.jpg" alt="" />
```

Vous pouvez inclure des fichiers images depuis un plugin chargé en utilisant *plugin syntax*. Pour inclure `app/Plugin/DebugKit/webroot/img/icon.png`, vous pouvez faire cela :

```
echo $this->Html->image('DebugKit.icon.png');
```

Si vous voulez inclure un fichier image qui partage un nom avec un plugin chargé vous pouvez faire ce qui suit. Par exemple si vous avez in plugin *Blog*', et voulez inclure également `app/webroot/js/Blog.icon.png` :

```
echo $this->Html->image('Blog.icon.png', array('plugin' => false));
```

Modifié dans la version 2.1 : L'option `fullBase` a été ajouté. Le support de *syntaxe de plugin* a été ajouté.

`HtmlHelper::link`(*string $title*, *mixed $url = null*, *array $options = array()*, *string $confirmMessage = false*)

param string $title Le texte à afficher comme corps du lien.

param mixed $url Soit la chaîne spécifiant le chemin, ou un *tableau routing*.

param array $options Un tableau d'attributs :*html attributes*.

Méthode générale pour la création de liens HTML. Utilisez les `$options` pour spécifier les attributs des éléments et si le `$title` devra ou non être échappé. :

```
echo $this->Html->link('Enter', '/pages/home', array('class' => 'button', 'target'
```

Affichera :

```
<a href="/pages/home" class="button" target="_blank">Enter</a>
```

Utilisez l'option `'full_base' => true` pour des URLs absolues :

```
echo $this->Html->link(
    'Dashboard',
    array('controller' => 'dashboards', 'action' => 'index', 'full_base' => true)
);
```

Affichera :

```
<a href="http://www.yourdomain.com/dashboards/index">Dashboard</a>
```

Spécifiez `$confirmMessage` pour afficher une boite de dialogue de confirmation `confirm()` javascript :

```
echo $this->Html->link(
    'Delete',
    array('controller' => 'recipes', 'action' => 'delete', 6),
    array(),
    "Are you sure you wish to delete this recipe?"
);
```

Affichera :

```
<a href="/recipes/delete/6" onclick="return confirm('Are you sure you wish to dele
```

Les chaînes de requête peuvent aussi être créées avec `link()`. :

```
echo $this->Html->link('View image', array(
    'controller' => 'images',
    'action' => 'view',
    1,
    '?' => array('height' => 400, 'width' => 500))
);
```

Affichera :

```
<a href="/images/view/1?height=400&width=500">View image</a>
```

Les caractères spéciaux HTML de `$title``seront convertis en entités HTML. Pour désactiver cette conversion, définissez l'option escape à false dans le tableau ``$options.`:

```
<?php
echo $this->Html->link(
    $this->Html->image("recipes/6.jpg", array("alt" => "Brownies")),
    "recipes/view/6",
    array('escape' => false)
);
```

Affichera :

```
<a href="/recipes/view/6">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Définir `escape` à false va aussi désactiver l'échappement des attributs du lien. Puisque depuis 2.4, vous pouvez utiliser l'option `escapeTitle` pour juste désactiver l'échappement du titre et pas des attributs. :

```
<?php
echo $this->Html->link(
    $this->Html->image('recipes/6.jpg', array('alt' => 'Brownies')),
    'recipes/view/6',
    array('escapeTitle' => false, 'title' => 'hi "howdy"')
);
```

Affichera :

```
<a href="/recipes/view/6" title="hi &quot;howdy&quot;">
    <img src="/img/recipes/6.jpg" alt="Brownies" />
</a>
```

Modifié dans la version 2.4 : L'option `escapeTitle` a été ajoutée. Regardez aussi la méthode `HtmlHelper::url` pour plus d'exemples des différents types d'urls.

`HtmlHelper::media`(*string|array $path*, *array $options*)

Paramètres

- **$path** (*string|array*) – Chemin du fichier vidéo, relatif au répertoire *webroot/{$options['pathPrefix']}*. Ou un tableau ou chaque élément peut être la chaîne d'un chemin ou un tableau associatif contenant les clés *src* et *type*.
- **$options** (*array*) – Un tableau d'attributs HTML, et d'options spéciales.
  Options :
  - *type* Type d'éléments média à générer, les valeurs valides sont "audio" ou "video". Si le type n'est pas fourni le type de média se basera sur le mime type du fichier.
  - *text* Texte à inclure dans la balise vidéo.
  - *pathPrefix* Préfixe du chemin à utiliser pour les urls relatives, par défaut à 'files/'.
  - *fullBase* Si il est fourni l'attribut src prendra l'adresse complète incluant le nom de domaine.

Introduit dans la version 2.1. Retourne une balise formatée audio/video :

```
.. code-block:: php

<?php echo $this->Html->media('audio.mp3'); ?>

// Affichera
<audio src="/files/audio.mp3"></audio>

<?php echo $this->Html->media('video.mp4', array(
    'fullBase' => true,
    'text' => 'Texte de remplacement'
)); ?>

// Affichera
<video src="http://www.somehost.com/files/video.mp4">Texte de remplacement</video>

<?php echo $this->Html->media(
    array('video.mp4', array('src' => 'video.ogg', 'type' => "video/ogg; codecs='theor
    array('autoplay')
); ?>

// Affichera
<video autoplay="autoplay">
    <source src="/files/video.mp4" type="video/mp4"/>
    <source src="/files/video.ogg" type="video/ogg; codecs='theora, vorbis'"/>
</video>
```

`HtmlHelper::tag`(*string $tag*, *string $text*, *array $htmlAttributes*)

Paramètres

- **$tag** (*string*) – Le nom de la balise créée.
- **$text** (*string*) – Le contenu de la balise.
- **$options** (*array*) – Un tableau d'attributs html *attributs html*.

Retourne des textes enveloppé dans une balise spécifiée. Si il n'y a pas de texte spécifié alors le contenu du <tag> sera retourné :

```
.. code-block:: php
```

< ?php echo $this->Html->tag('span', 'Bonjour le Monde', array('class' => 'welcome')) ; ?>

// Affichera Bonjour le Monde

// Pas de texte spécifié. < ?php echo $this->Html->tag('span', null, array('class' => 'welcome')) ; ?>

// Affichera

Note : Le texte n'est pas échappé par défaut mais vous pouvez utiliser `$htmlOptions['escape'] = true` pour échapper votre texte. Ceci remplace un quatrième paramètre `boolean $escape = false` qui était présent dans les précédentes versions.

`HtmlHelper::div`(*string $class*, *string $text*, *array $options*)

Paramètres

- **$class** (*string*) – Le nom de classe de la div.
- **$text** (*string*) – Le contenu de la div.
- **$options** (*array*) – Un tableau d'attributs *attributs html*.

Utilisé pour les sections de balisage enveloppés dans des div. Le premier paramètre spécifie une clasee CSS, et le second est utilisé pour fournir le texte à envelopper par les balises div. Si le dernier paramètre à été défini à true, $text sera affiché en HTML-échappé.

Si aucun texte n'est spécifié, seulement une balise div d'ouverture est retournée :

```
.. code-block:: php
```

< ?php echo $this->Html->div('error', 'Entrez votre numéro de carte bleue S.V.P') ; ?>

// Affichera Entrez votre numéro de carte bleue S.V.P

`HtmlHelper::para`(*string $class*, *string $text*, *array $options*)

Paramètres

- **$class** (*string*) – Le nom de classe du paragraphe.
- **$text** (*string*) – Le contenu du paragraphe.
- **$options** (*array*) – Un tableau d'attributs *attributs html*.

Retourne un texte enveloppé dans une balise CSS . Si aucun texte CSS est fourni, un simple de démarrage est retourné :

```
.. code-block:: php
```

< ?php echo $this->Html->para(null, 'Bonjour le Monde') ; ?>

// Affichera Bonjour le Monde

`HtmlHelper::script` (*mixed $url*, *mixed $options*)

Paramètres

- **$url** (*mixed*) – Soit un simple fichier Javascript, ou un tableau de chaînes pour plusieurs fichiers.
- **$options** (*array*) – Un tableau d'attributs *attributs html*.

Inclus un(des) fichier(s), présent soit localement soit à une url distante.

Par défaut, les tags de script sont ajoutés au document inline. Si vous le surcharger en configurant `$options['inline']` à false, les tags de script vont plutôt être ajoutés au block `script` que vous pouvez afficher aileurs dans le document. Si vous souhaitez surcharger le nom du block utilisé, vous pouvez le faire en configurant `$options['block']`.

`$options['once']` contrôle si vous voulez ou pas inclure le script une fois par requête. Par défaut à true.

Vous pouvez utiliser $options pour définir des propriétés supplémentaires pour la balise script générée. Si un tableau de balise script est utilisé, les attributs seront appliqués à toutes les balises script générées.

Cette méthode d'inclusion de fichier javascript suppose que les fichiers javascript spécifiés se trouvent dans le répertoire `/app/webroot/js`. :

```
echo $this->Html->script('scripts');
```

Affichera :

```
<script type="text/javascript" href="/js/scripts.js"></script>
```

Vous pouvez lier à des fichiers avec des chemins absolus tant qu'ils ne se trouvent pas dans `app/webroot/js` :

```
echo $this->Html->script('/autrerep/fichier_script');
```

Vous pouvez aussi lier à une URL d'un dépôt distant :

```
echo $this->Html->script('http://code.jquery.com/jquery.min.js');
```

Affichera :

```
<script type="text/javascript" href="http://code.jquery.com/jquery.min.js"></script>
```

Le premier paramètre peut être un tableau pour inclure des fichiers multiples. :

```
echo $this->Html->script(array('jquery', 'wysiwyg', 'scripts'));
```

Affichera :

```
<script type="text/javascript" href="/js/jquery.js"></script>
<script type="text/javascript" href="/js/wysiwyg.js"></script>
<script type="text/javascript" href="/js/scripts.js"></script>
```

Vous pouvez insérer dans la balise script un bloc spécifique en utilisant l'option `block`. :

```
echo $this->Html->script('wysiwyg', array('block' => 'scriptBottom'));
```

Dans votre layout, vous pouvez ressortir toutes les balises script ajoutées dans 'scriptBottom' :

```
echo $this->fetch('scriptBottom');
```

Vous pouvez inclure des fichiers de script depuis un plugin en utilisant la syntaxe *syntaxe de plugin*. Pour inclure `app/Plugin/DebugKit/webroot/js/toolbar.js` vous devriez faire cela :

```
echo $this->Html->script('DebugKit.toolbar.js');
```

Si vous voulez inclure un fichier de script qui partage un nom de fichier avec un plugin chargé vous pouvez faire cela. Par exemple si vous avez Un plugin `Blog`, et voulez inclure également `app/webroot/js/Blog.plugins.js`, vous devriez :

```
echo $this->Html->script('Blog.plugins.js', array('plugin' => false));
```

Modifié dans la version 2.1 : L'option `block` a été ajouté. Le support de la syntaxe *syntaxe de plugin* a été ajouté.

`HtmlHelper::`**`scriptBlock`**(*$code*, *$options = array()*)

Paramètres

- **$code** (*string*) – Le code à placer dans la balise script.
- **$options** (*array*) – Un tableau d'attributs *attributs html*.

Génère un bloc de code contenant des options `$options['inline']` définies de `$code` à mettre à false pour voir le bloc de script apparaître dans le bloc de `script` de la vue. D'autre options définies seront ajoutée comme attributs dans les balises de script. `$this->Html->scriptBlock('stuff', array('defer' => true));` créera une balise script avec l'attribut `defer="defer"`.

`HtmlHelper::`**`scriptStart`**(*$options = array()*)

Paramètres

- **$options** (*array*) – Un tableau d'attributs *html attributes* à utiliser quand scriptEnd est appelé.

Débute la mise en mémoire tampon d'un bloc de code. Ce bloc de code va capturer toutes les sorties entre `scriptStart()` et `scriptEnd()` et crée une balise script. Les options sont les mêmes que celles de `scriptBlock()`

`HtmlHelper::`**`scriptEnd`**()

Termine la mise en mémoire tampon d'un bloc de script, retourne l'élément script généré ou null si le bloc de script à été ouvert avec inline=false.

Un exemple de l'utilisation de `scriptStart()` et `scriptEnd()` pourrait être :

```
$this->Html->scriptStart(array('inline' => false));

echo $this->Js->alert('je suis dans le javascript');

$this->Html->scriptEnd();
```

`HtmlHelper::`**`nestedList`**(*array $list*, *array $options = array()*, *array $itemOptions = array()*, *string $tag = 'ul'*)

Paramètres

- **$list** (*array*) – Ensemble d'éléments à lister.
- **$options** (*array*) – Attributs HTML supplémentaires des balises de listes (ol/ul) ou si ul/ol utilise cela comme une balise.

- **$itemOptions** (*array*) – Attributs additionnels des balises de listes item(LI).
- **$tag** (*string*) – Type de balise liste à utiliser (ol/ul).

Fabrique une liste imbriquée (UL/OL) dans un tableau associatif :

```
$list = array(
    'Languages' => array(
        'English' => array(
            'American',
            'Canadian',
            'British',
        ),
        'Spanish',
        'German',
    )
);
echo $this->Html->nestedList($list);
```

Sortie :

```
// Affichera (sans les espaces blancs)
<ul>
    <li>Languages
        <ul>
            <li>English
                <ul>
                    <li>American</li>
                    <li>Canadian</li>
                    <li>British</li>
                </ul>
            </li>
            <li>Spanish</li>
            <li>German</li>
        </ul>
    </li>
</ul>
```

`HtmlHelper::tableHeaders`(*array $names*, *array $trOptions = null*, *array $thOptions = null*)

Paramètres

- **$names** (*array*) – Un tableau de chaîne pour créé les entête de tableau.
- **$trOptions** (*array*) – Un tableau d'attributs *html attributes* pour le <tr>.
- **$thOptions** (*array*) – Un tableau d'attributs *html attributes* pour l'élément <th>.

Créé une ligne de cellule d'en-tête à placer dans la balise <table>. :

```
echo $this->Html->tableHeaders(array('Date', 'Title', 'Active'));
```

// Affichera

```
<tr>
    <th>Date</th>
    <th>Title</th>
    <th>Active</th>
</tr>
```

```
echo $this->Html->tableHeaders(
    array('Date','Title','Active'),
    array('class' => 'status'),
    array('class' => 'product_table')
);
```

Sortie :

```
<tr class="status">
     <th class="product_table">Date</th>
     <th class="product_table">Title</th>
     <th class="product_table">Active</th>
</tr>
```

Modifié dans la version 2.2 : `tableHeaders()` accepte maintenant les attributs par cellule, regardez ci-dessous. Depuis 2.2 vous pouvez définir des attributs par colonne, ceux-ci sont utilisés à la place de ceux par défaut dans `$thOptions` :

```
echo $this->Html->tableHeaders(array(
    'id',
    array('Name' => array('class' => 'highlight')),
    array('Date' => array('class' => 'sortable'))
));
```

Sortie :

```
<tr>
    <th>id</th>
    <th class="highlight">Name</th>
    <th class="sortable">Date</th>
</tr>
```

`HtmlHelper::tableCells`(*array $data*, *array $oddTrOptions = null*, *array $evenTrOptions = null*, *$useCount = false*, *$continueOddEven = true*)

Paramètres

- **$data** (*array*) – Un tableau à deux dimensions avec les données pour les lignes.
- **$oddTrOptions** (*array*) – Un tableau d'attributs *html attributes* pour les <tr> impairs.
- **$evenTrOptions** (*array*) – Un tableau d'attributs *html attributes* pour les <tr> pairs.
- **$useCount** (*boolean*) – Ajoute la classe "column-$i".
- **$continueOddEven** (*boolean*) – Si à false, utilisera une variable $count non-statique, ainsi le compteur impair/pair est remis à zéro juste pour cet appel.

Créé des cellules de table, en assignant aux lignes des attributs <tr> différents pour les lignes paires et les lignes impaires. Entoure une simple table de cellule dans un array() pour des attributs <td> spécifiques.

```
echo $this->Html->tableCells(array(
    array('Jul 7th, 2007', 'Best Brownies', 'Yes'),
    array('Jun 21st, 2007', 'Smart Cookies', 'Yes'),
    array('Aug 1st, 2006', 'Anti-Java Cake', 'No'),
));
```

Sortie :

```
<tr><td>Jul 7th, 2007</td><td>Best Brownies</td><td>Yes</td></tr>
<tr><td>Jun 21st, 2007</td><td>Smart Cookies</td><td>Yes</td></tr>
<tr><td>Aug 1st, 2006</td><td>Anti-Java Cake</td><td>No</td></tr>
```

```
echo $this->Html->tableCells(array(
    array('Jul 7th, 2007', array('Best Brownies', array('class' => 'highlight')) , 'Ye
    array('Jun 21st, 2007', 'Smart Cookies', 'Yes'),
    array('Aug 1st, 2006', 'Anti-Java Cake', array('No', array('id' => 'special'))),
));
```

// Sortie

```
<tr><td>Jul 7th, 2007</td><td class="highlight">Best Brownies</td><td>Yes</td></tr>
<tr><td>Jun 21st, 2007</td><td>Smart Cookies</td><td>Yes</td></tr>
<tr><td>Aug 1st, 2006</td><td>Anti-Java Cake</td><td id="special">No</td></tr>
```

```
echo $this->Html->tableCells(
    array(
        array('Red', 'Apple'),
        array('Orange', 'Orange'),
        array('Yellow', 'Banana'),
    ),
    array('class' => 'darker')
);
```

Output :

```
<tr class="darker"><td>Red</td><td>Apple</td></tr>
<tr><td>Orange</td><td>Orange</td></tr>
<tr class="darker"><td>Yellow</td><td>Banana</td></tr>
```

`HtmlHelper::url` (*mixed $url = NULL*, *boolean $full = false*)

Paramètres

- **$url** (*mixed*) – Un tableau de routing *routing array*.
- **$full** (*mixed*) – Soit un booléen s'il faut ou pas que la base du chemin soit incluse ou un tableau d'options pour le router `Router::url()`.

Retourne une URL pointant vers une combinaison controller et action. Si $url est vide, cela retourne la REQUEST_URI, sinon cela génère la combinaison d'une url pour le controller et d'une action. Si full est à true, la base complète de l'URL sera ajoutée en amont du résultat :

```
echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "view",
    "bar"
));

// Restituera
/posts/view/bar
```

Voici quelques exemples supplémentaires :

URL avec des paramètres nommés :

```
echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "view",
    "foo" => "bar"
));

// Restituera
/posts/view/foo:bar
```

URL avec une extension :

```
echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "list",
    "ext" => "rss"
));

// Restituera
/posts/list.rss
```

URL (commençant par '/') avec la base complète d'URL ajoutée :

```
echo $this->Html->url('/posts', true);

// Restituera
http://somedomain.com/posts
```

URL avec des paramètres GET et une ancre nommée :

```
<?php echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "search",
    "?" => array("foo" => "bar"),
    "#" => "first"));

// Restituera
/posts/search?foo=bar#first
```

Pour plus d'information voir Router : :url (http ://api20.cakephp.org/class/router#method-Routerurl) dans l' API.

`HtmlHelper::useTag`(*string $tag*)

Retourne un bloc existant formaté de balise `$tag` :

```
$this->Html->useTag(
    'form',
    'http://example.com',
    array('method' => 'post', 'class' => 'myform')
);
```

Output :

```
<form action="http://example.com" method="post" class="myform">
```

Changer la restitution des balises avec le Helper Html

`HtmlHelper::loadConfig` (*mixed $configFile*, *string $path = null*)

Les jeux de balises pour le Helper Html `HtmlHelper` sont conforme au standard XHTML, toutefois si vous avez besoin de générer du HTML pour les standards HTML5 vous aurez besoin de créer et de charger un nouveau fichier de configuration de balise contenant les balises que vous aimeriez utiliser. Pour changer les balises utilisées créez un fichier `app/Config/html5_tags.php` contenant :

```
$config = array('tags' => array(
    'css' => '<link rel="%s" href="%s" %s>',
    'style' => '<style%s>%s</style>',
    'charset' => '<meta charset="%s">',
    'javascriptblock' => '<script%s>%s</script>',
    'javascriptstart' => '<script>',
    'javascriptlink' => '<script src="%s"%s></script>',
    // ...
));
```

Vous pouvez alors charger ces balises définis en appelant `$this->Html->loadConfig('html5_tags');`.

Création d'un chemin de navigation avec le Helper Html

`HtmlHelper::getCrumbs` (*string $separator = '&raquo ;'*, *string $startText = false*)

CakePHP inclut la possibilité de créer automatiquement un chemin de navigation (fil d'Ariane) dans votre application. Pour mettre cela en service, ajouter cela dans votre template de layout :

```
echo $this->Html->getCrumbs(' > ', 'Home');
```

L'option `$startText` peut aussi acceptée un tableau. Cela donne plus de contrôle à travers le premier lien généré :

```
echo $this->Html->getCrumbs(' > ', array(
    'text' => $this->Html->image('home.png'),
    'url' => array('controller' => 'pages', 'action' => 'display', 'home'),
    'escape' => false
));
```

Une clé qui n'est pas `text` ou `url` sera passée à `link()` comme paramètre `$options`. Modifié dans la version 2.1 : Le paramètre `$startText` accepte maintenant un tableau.

`HtmlHelper::addCrumb` (*string $name*, *string $link = null*, *mixed $options = null*)

Maintenant, dans votre vue vous allez devoir ajouter ce qui suit pour démarrer le fil d'Ariane sur chacune de vos pages. :

```
$this->Html->addCrumb('Users', '/users');
$this->Html->addCrumb('Add User', '/users/add');
```

Ceci ajoutera la sortie "**Home > Users > Add User**" dans votre layout où le fil d'Ariane a été ajouté.

`HtmlHelper::getCrumbList` (*array $options = array()*, *mixed $startText*)

Paramètres

- **$options** (*array*) – Un tableau de *html attributes* pour les elements contenant `<ul>`. Peut aussi contenir les options 'separator', 'firstClass' et 'lastClass'.
- **$startText** (*string|array*) – Le texte ou l'elément qui précède ul.

Retourne le fil d'Ariane comme une liste (x)html.

Cette méthode utilise `HtmlHelper::tag()` pour générer la liste et ces éléments. Fonctionne de la même manière que `getCrumbs()`, il utilise toutes les options que chacun des fils a ajouté. Vous pouvez utiliser le paramètre `$startText` pour fournir le premier lien de fil. C'est utile quand vous voulez inclure un lien racine. Cette option fonctionne de la même façon que l'option `$startText` pour `getCrumbs()`. Modifié dans la version 2.1 : Le paramètre `$startText` a été ajouté.Modifié dans la version 2.3 : Les options 'separator', 'firstClass' et 'lastClass' ont été ajoutées.

JSHelper

`Helper Js(View $view, array $settings = array())`

Depuis le début le support de CakePHP pour Javascript a été orienté vers Prototype/Scriptaculous (librairie javascript). Tandis que nous continuons de penser qu'il s'agit d'une excellente bibliothèque Javascript, il a été demandé à la communauté de supporter d'autres librairies. Plutôt que d'enlever Prototype en faveur d'une autre librairie javascript. Nous avons crées un adaptateur fonctionnant sur le principe d'un Helper et avons inclu 3 des librairies les plus demandées. Prototype/scriptaculous, Mootools/Mootools-more, et jQuery/jQueryUI. Et bien que l'API n'est pas aussi vaste que le Helper Ajax, nous pensons que la solution basée sur l'adaptateur permet une solution plus extensible offrant aux développeurs la puissance et la flexibilité dont ils ont besoin pour répondre à leurs besoins spécifiques.

Les moteurs Javascript forment l'épine dorsale du nouveau Helper Js. Un moteur javascript traduit un élément Javascript abstrait dans un code javascript concret spécifique à la librairie en cours d'utilisation. De plus ils créent un système extensible à utiliser pour les autres.

Utilisation d'un moteur Javascript spécifique

Avant tout, téléchargez votre librairie javascript préférée et placez la dans `app/webroot/js`.

Puis, vous devez inclure la librairie dans votre page. Pour l'inclure dans toutes les pages, ajoutez cette ligne dans la section <head> de `app/View/Layouts/default.ctp` (copiez ce fichier depuis `lib/Cake/View/Layouts/default.ctp` si vous n'avez pas créé le votre) :

```
echo $this->Html->script('jquery'); // Inclut la librairie Jquery
```

Remplacez `jquery` par le nom de votre fichier de librairie (.js sera ajouté au nom).

Par défaut les scripts sont mis en cache, et vous devez explicitement imprimer le cache. Pour faire cela à la fin de chacune des pages, incluez cette ligne juste avant la balise de fin de `</body>`

```
echo $this->Js->writeBuffer(); // Écrit les scripts en mémoire cache
```

Attention : Vous devez inclure la librairie dans votre page et afficher le cache pour que le helper fonctionne.

La selection du moteur Javascript est déclarée quand vous incluez le helper dans votre controller

```
public $helpers = array('Js' => array('Jquery'));
```

La partie ci-dessus utilise le moteur Jquery dans les instances du Helper Js dans vos vues. Si vous ne déclarez pas un moteur spécifique, le moteur Jquery sera utilisé par défaut. Comme il est mentionné ci-dessus, il y a trois moteurs implémentés dans le noyau, mais nous encourageons la communauté à étendre la compatibilité des librairies.

Utilisation de jQuery avec d'autres librairies La librairie jQuery, et virtuellement tous ses plugins sont limités au sein de l'espace jQuery. Comme règle générale, les objets "globaux" sont stockés dans l'espace JQuery, ainsi vous ne devriez pas avoir de clash entre Jquery et d'autre librairies (comme Prototype, MooTools, ou YUI).

Ceci dit, il y a une mise en garde : **Par défaut, jQuery utilise "$" comme raccourci de "jQuery"**

Pour redéfinir le raccourci "$", utilisez la variable jQueryObject :

```
$this->Js->JqueryEngine->jQueryObject = '$j';
echo $this->Html->scriptBlock(
    'var $j = jQuery.noConflict();',
    array('inline' => false)
);
// Demande à jQuery de se placer dans un mode noconflict
```

Utilisation du Helper Js dans des helpers personnalisés Déclarez le Helper Js dans le tableau `$helpers` de votre Helper personnalisé :

```
public $helpers = array('Js');
```

Note : Il n'est pas possible de déclarer le moteur javascript dans un Helper personnalisé. Ceci n'aurait aucun effet.

Si vous êtes prêt à utiliser un moteur javascript autre que celui par défaut, faîtes le paramétrage du Helper dans votre controller comme ceci :

```
public $helpers = array(
    'Js' => array('Prototype'),
    'CustomHelper'
);
```

> **Warning :** Assurez-vous de déclarer le Helper Js et son moteur **en haut** du tableau `$helpers` dans votre controller.

Le moteur javascript séléctionné peut disparaître (remplacé par celui par défaut) de l'objet Helper Js dans votre helper, si vous oubliez de faire cela et vous obtiendrez du code qui ne correspond pas à votre librairie javascript.

Création d'un moteur Javascript

Les helpers de moteur Javascript suivent les conventions normales des helper, avec quelques restrictions supplémentaires. Ils doivent avoir le suffixe `Engine`. `DojoHelper` n'est pas bon, `DojoEngineHelper` est correct. De plus ils doivent étendre `JsBaseEngineHelper` afin de tirer parti du meilleur de la nouvelle API.

Utilisation du moteur Javascript

Le `JsHelper` fournit quelques méthodes, et agit comme une façade pour le moteur helper. Vous ne devriez pas accéder au moteur helper excepté dans de rares occasions. Utilisez les fonctionnalités de façade du `Helper Js` vous permet de tirer parti de la mise en mémoire tampon et de la méthode caractéristique de chaînage intégré ; (le chaînage de méthode ne fonctionne que dans PHP5).

Par défaut le `Helper Js` bufferise presque tous les codes du script générés, ce qui vous permet de récupérer les scripts partout dans la vue, les éléments et les mises en page, et de les ressortir à un endroit. La Récupération des scripts bufferisés est réalisé avec `$this->Js->writeBuffer();` ceci retournera le contenu du buffer dans une balise script. Vous pouvez désactiver le buffering généralisé avec la propriété `$bufferScripts` ou en définissant `buffer => false` dans les méthodes qui prennent des `$options`.

Étant donné que la plupart des méthodes en Javascript commencent avec une sélection d'éléments dans le DOM, `$this->Js->get()` retourne un $this, vous permettent d'enchaîner les méthodes en utilisant la selection. Le chaînage de méthode vous permet d'écrire moins, et de rendre votre code plus expressif . :

```
$this->Js->get('#foo')->event('click', $eventCode);
```

Est un exemple de chaînage de méthode. Le chaînage de méthode n'est pas possible dans PHP4 et l'exemple ci-dessus devrait être écrit comme :

```
$this->Js->get('#foo');
$this->Js->event('click', $eventCode);
```

Options communes Dans le but de simplifier le développement et comme les librairies Js peuvent changer, un ensemble courant d'options est pris en charge par `JsHelper`, Ces options courantes seront mappées en dehors des options spécifiques de la librairies en interne. Si vous ne prévoyez pas la commutation des librairies, chaque librairie supporte toutes les fonctions de callback natives et les options.

Enveloppement de Callback Par défaut, toutes les options de callback sont enveloppées dans une fonction anonyme avec les bons arguments. Vous pouvez désactiver ce comportement en fournissant `wrapCallbacks = false` dans votre tableau d'options.

Travailler avec des scripts bufferisés Un inconvénient à la précédente implémentation des fonctionnalités de type d'Ajax était la dispersion des balises de script partout dans le document , et l'impossibilité de bufferiser les scripts ajoutés par les éléments dans la mise en page. Le nouveau Helper Js si il est utilisé correctement évite ces deux questions. Il est recommandé de placer `$this->Js->writeBuffer()` à

la fin du fichier layout au dessus la balise `</body>`. Ceci permettra à tous les scripts générés dans les éléments du layout d'être ressortis (output) à un endroit. Il doit être noté que les scripts bufferisés sont gérés séparément des scripts de fichiers inclus.

writeBuffer(*$options = array()*)

Écrit tous le codes Javascript générés jusqu'ici dans un bloc de code ou les met en mémoire cache dans un fichier et retourne un script lié.

Options
- `inline` - Défini à true pour avoir la sortie des scripts dans un bloc de script inline. si cache est aussi à true, une balise de lien de script sera générée (par défaut à true)
- `cache` - Défini à true pour avoir les scripts dans un fichier de la mémoire cache et s'y relié (false par défaut)
- `clear` - Défini à false pour éviter au fichier de cache d'être effacé (true par défaut)
- `onDomReady` - enveloppe les scripts en mémoire cache dans un evénement domready (par défaut à true)
- `safe` - si un block inline est généré il sera enveloppé dans < ![CDATA[...]]> (true par défaut)

La création d'un fichier de cache avec `writeBuffer()` nécessite que `webroot/js` soit accessible en écriture et permette au navigateur de placer dans le cache les ressources de script généré pour la page.

buffer(*$content*)

Ajoute `$content` au buffer de script interne.

getBuffer(*$clear = true*)

Récupère le contenu du buffer courant. Passe false pour ne pas effacer le buffer en même temps.

Bufferiser des méthodes qui ne sont normalement pas bufferisée

Quelques méthodes dans le Helper sont bufferisée par défaut. Le moteur bufferise les méthodes suivantes par défaut :
- event
- sortable
- drag
- drop
- slider

De plus vous pouvez forcer une autre méthode du Helper Js à utiliser la mise en mémoire cache. En ajoutant un booléen à la fin des arguments vous pouvez forcer d'autres méthodes d'aller en mémoire cache. Par exemple la méthode `each()` qui n'est normalement pas bufferisée :

```
$this->Js->each('alert("sapristi!");', true);
```

Ce qui est ci-dessus va forcer la méthode `each()` à utiliser le buffer. En revanche si vous souhaitez qu'une méthode bufferisée ne bufferise plus, vous pouvez passer un `false` comme le dernier argument :

```
$this->Js->event('click', 'alert("sapristi!");', false);
```

Ceci forcera la fonction event qui est normalement mis en mémoire cache à retourner son résultat.

D'autres Méthodes

Les moteurs Javascript du noyau fournissent les mêmes fonctionnalités définies a travers les autres librairies, il y a aussi un sous-ensemble d'options communes qui sont traduites dans les options spécifiques des librairies. Tout cela pour fournir au développeurs finaux une Api unifiée autant que possible. La liste suivante de méthodes est supportée par tous les moteurs inclus dans le noyau CakePHP. Chaque fois que vous voyez une liste séparée pour les `Options` et les `Event Options` Les deux ensembles de paramètres sont fournis dans le tableau `$options` pour la méthode.

object (*$data*, *$options = array()*)

Sérialise `$data` vers JSON. Cette méthode est un proxy pour `json_encode()` avec quelques fonctionnalités supplémentaires ajoutée avec le paramètre `$options`.

Options :

–`prefix` - Chaîne ajoutée en début des données retournées.

–`postfix` - Chaîne ajoutée aux donnée retournée.

Exemple d'utilisation :

```
$json = $this->Js->object($data);
```

sortable (*$options = array()*)

Sortable génère un extrait de code pour fabriquer un ensemble d'éléments (souvent une liste) drag and drop triable.

Les options normalisées sont :

Options

–`containment` - Conteneur de l'action de déplacement.

–`handle` - Selecteur de l'élement. Seul cet élément commencera l'action de tri.

–`revert` - S'il faut ou pas utiliser un effet pour déplacer l'élément triable dans sa position finale.

–`opacity` - Opacité de l'espace réservé.

–`distance` - Distance a laquelle l'élément triable doit être draggé avant que le tri n'opère.

Event Options

–`start` - Événement lancé quand le tri commence.

–`sort` - Événement lancé quand le tri est en cours.

–`complete` - Événement lancé quand le tri est terminé.

D'autres options sont supportées par chacune des librairies Javascript, et vous pouvez obtenir dans leurs documentation respective des informations plus détaillées sur les options et les paramètres.

Example d'utilisation :

```
$this->Js->get('#ma-liste');
$this->Js->sortable(array(
    'distance' => 5,
    'containment' => 'parent',
    'start' => 'onStart',
    'complete' => 'onStop',
    'sort' => 'onSort',
    'wrapCallbacks' => false
));
```

En imaginant que vous étiez en train d'utiliser le moteur Jquery, vous devriez avoir le code suivant dans votre block Javascript généré.

```
$("#maListe").sortable({containment:"parent", distance:5, sort:onSort, start:onStart,
```

request (*$url*, *$options = array()*)

Génère un morceau de code Javascript pour créer une requête `XmlHttpRequest` ou 'AJAX'.

Options de l'événement

–`complete` - Callback à lancer si complété.
–`success` - Callback à lancer en cas de succès.
–`before` - Callback à lancer à l'initialisation de la requête.
–`error` - Callback à lancer en cas d'erreur de requête.

Options

–`method` - La méthode pour fabriquer la requête avec GET dans plus de librairies.
–`async` - S'il faut ou pas utiliser une requête asynchrone.
–`data` - Données additionnelles à envoyer.
–**`update` - L'ID du Dom id à mettre à jour avec le contenu de la** requête.
–`type` - Le Type des données de la réponse.'json' et 'html' sont supportés. Par défaut à html pour la plupart des librairies.
–`evalScripts` - s'il faut ou pas évaluer la balise <script>.
–**`dataExpression` -Si la clef `data` doit être traitée comme un** callback. Utile pour fournir `$options['data']` comme une autre expression Javascript.

Exemple d'utilisation :

```
$this->Js->event(
    'click',
    $this->Js->request(
        array('action' => 'foo', 'param1'),
        array('async' => true, 'update' => '#element')
    )
);
```

get (*$selector*)

Définit la 'sélection' interne dans un sélecteur CSS. La sélection active est utilisée dans les opérations ultérieures jusqu'à ce qu'une nouvelle soit faite. :

```
$this->Js->get('#element');
```

Le `JsHelper` fait maintenant référence à toutes les méthodes de la sélection basées sur #element. Pour changer la sélection active appelez `get()` à nouveau avec un nouvel élément.

set (*mixed $one*, *mixed $two = null*)

Passe des variables dans javascript. Vous permet de définir des variables qui seront retournées quand le buffer est extrait avec `Helper Js::getBuffer()` ou `Helper Js::writeBuffer()`. La variable Javascript utilisée pour retourner les variables peut être controllée avec `Helper Js::$setVariable`.

drag (*$options = array()*)

Rend un élément draggable.

Options

–`handle` - selecteur de l'élement.
–`snapGrid` - La grille de pixel qui déclenche les mouvements, un tableau(x, y)

–**container - L'élément qui agit comme un rectangle de selection pour** l'élément draggable.

Options d'événements

–start - Événement lancé quand le drag démarre.

–drag - Événement lancé à chaque étape du drag.

–stop - Événement lancé quand le drag s'arrête. (souris relâchée)

Exemple d'utilisation :

```
$this->Js->get('#element');
$this->Js->drag(array(
    'container' => '#content',
    'start' => 'onStart',
    'drag' => 'onDrag',
    'stop' => 'onStop',
    'snapGrid' => array(10, 10),
    'wrapCallbacks' => false
));
```

Si vous utilisiez le moteur Jquery le code suivant devrait être ajouté au buffer

```
$("#element").draggable({containment:"#content", drag:onDrag, grid:[10,10], start:
```

drop(*$options = array()*)

Fabrique un élément accepte des éléments dragguables et agit comme dropzone pour les éléments draggés.

Options

–accept - Sélecteur des éléments que ce droppable acceptera.

–hoverclass - Classe pour ajouter à droppable quand un draggable est terminé.

Event Options

–drop - Événement lancé quand un élément est droppé dans la drop zone.

–hover - Événement lancé quand un drag entre dans une drop zone.

–leave - Événement lancé quand un drag est retiré depuis une drop zone sans être droppé.

Exemple d'utilisation :

```
$this->Js->get('#element');
$this->Js->drop(array(
    'accept' => '.items',
    'hover' => 'onHover',
    'leave' => 'onExit',
    'drop' => 'onDrop',
    'wrapCallbacks' => false
));
```

Si vous utilisiez le moteur jQuery le code suivant devrait être ajouté au buffer.

```
$("#element").droppable({accept:".items", drop:onDrop, out:onExit, over:onHover});
```

Note : Les éléments Droppables dans Mootools fonctionnent différemment des autres librairies. Les Droppables sont implémentés comme une extension de Drag. Donc pour faire une selection get() pour l'élément droppable. Vous devez aussi fournir une règle de selecteur à l'élément draggable. De plus, les droppables Mootools héritent de toutes les option de Drag.

slider (*$options = array()*)

Créé un morceau de code Javascript qui converti un élément dans un morceau de code slider ui. Voir les implémentations des différentes librairies pour des utilisations supplémentaires et les fonctionnalités.

Options

–`handle` - l' id de l'élément utilisé dans le sliding.
–`direction` - La direction du slider soit 'vertical' ou 'horizontal'.
–`min` - La valeur minimale pour le slider.
–`max` - La valeur maximale pour le slider.
–`step` - Le nombre d'étapes que le curseur aura.
–`value` - Le décalage initial du slider.

Events

–`change` - Lancé quand la valeur du slider est actualisé.
–`complete` - Lancé quand un utilisateur arrête de slider le gestionnaire.

Exemple d'utilisation :

```
$this->Js->get('#element');
$this->Js->slider(array(
    'complete' => 'onComplete',
    'change' => 'onChange',
    'min' => 0,
    'max' => 10,
    'value' => 2,
    'direction' => 'vertical',
    'wrapCallbacks' => false
));
```

Si vous utilisiez le moteur jQuery le code suivant devrait être ajouté au buffer.

```
$("#element").slider({change:onChange, max:10, min:0, orientation:"vertical", stop:onC
```

effect (*$name*, *$options = array()*)

Créé un effet basique. Par défaut cette méthode n'est pas bufferisée et retourne ses résultats.

noms des effets supportés

Les effets suivants sont supportés par tous les moteurs JS :
–`show` - révèle un élément.
–`hide` - dissimule un élément.
–`fadeIn` - Fade in un élément.
–`fadeOut` - Fade out un élément.
–`slideIn` - Slide un élément in.
–`slideOut` - Slide un élément out.

Options

–`speed` - Vitesse à laquelle l'animation devrait se produire. Les valeurs acceptées sont 'slow', 'fast'. Tous les effets n'utilisent pas l'option speed.

Exemple d'utilisation

Si vous utilisez le moteur jQuery :

```
$this->Js->get('#element');
$result = $this->Js->effect('fadeIn');

// $result contient $("#foo").fadeIn();
```

event (*$type*, *$content*, *$options = array()*)

Attache un événement à la sélection courante. `$type` peut être un événement DOM normal ou un type d'événement personnalisé si votre librairie les supporte. `$content` devrait contenir les fonctions du body pour le callback. Les Callbacks seront enveloppés avec la fonction `function (event) { ... }` à moins qu'ils ne soient désactivés avec `$options`.

Options

–`wrap` - Si vous souhaitez que le callback soit enveloppé dans une fonction anonyme. (par défaut à true)

–`stop` - Si vous souhaitez que l'événement s'arrête. (par défaut à true)

Exemple d'utilisation :

```
$this->Js->get('#some-link');
$this->Js->event('click', $this->Js->alert('saperlipopette!'));
```

Si vous employiez la librairie jQuery, vous devriez avoir le code suivant :

```
$('#some-link').bind('click', function (event) {
    alert(saperlipopette!');
    return false;
});
```

Vous pouvez retirer le `return false;` en passant l'option `stop` à false :

```
$this->Js->get('#some-link');
$this->Js->event('click', $this->Js->alert('saperlipopette!'), array('stop' => false))
```

Si vous employiez la librairie jQuery vous devriez avoir le code Javascript suivant ajouté au buffer. Notez que l'événement du navigateur par défaut n'est pas annulé :

```
$('#some-link').bind('click', function (event) {
    alert('hey you!');
});
```

domReady (*$callback*)

Créé l'événement spécial 'DOM ready'. `JsHelper::writeBuffer()` enveloppe automatiquement les scripts bufferisés dans une méthode domReady.

each (*$callback*)

Créé un morceau de code qui effectue une itération sur les éléments sélectionnés, et insère `$callback`.

Exemple :

```
$this->Js->get('div.message');
$this->Js->each('$(this).css({color: "red"});');
```

L'utilisation du moteur jQuery aurait créé le Javascript suivant :

```
$('div.message').each(function () { $(this).css({color: "red"}); });
```

alert (*$message*)

Créé un extrait de code javascript contenant un `alert()`. Par défaut, `alert` ne bufferise pas, et retourne le morceau de script suivant. :

```
$alert = $this->Js->alert('Zogotunga!');
```

confirm(*$message*)

Créé un bout de code contenant `confirm()`. Par défaut, `confirm` ne bufferise pas, et retourne le morceau de script suivant. :

```
$alert = $this->Js->confirm('Vraiment certain?');
```

prompt(*$message*, *$default*)

Créé un bout de code Javascript contenant `prompt()`. Par défaut, `prompt` ne bufferise pas, et retourne le morceau de code suivant. :

```
$prompt = $this->Js->prompt('C'est quoi ta couleur préférée?', 'bleu');
```

submit(*$caption = null*, *$options = array()*)

Créé un bouton submit qui permet les formulaires de soumission `XmlHttpRequest`. Les options peuvent inclure soit celles de `FormHelper::submit()` et JsBaseEngine : :request(), JsBaseEngine : :event() ;

La soumission a travers un formulaire avec cette méthode, ne permet pas l'envoi de fichiers. Les fichiers ne se transferts pas à travers `XmlHttpRequest` et requièrent un iframe, ou d'autres paramétrages plus spécialisés qui sont hors de portée de cet helper.

Options

–`confirm` - Message de confirmation affiché avant l'envoi de la requête. L'utilisation de 'confirm' , ne remplace pas les méthodes de callback `before` dans le XmlHttpRequest généré.

–`buffer` - Désactive le buffering et retourne une balise script en plus du lien.

–`wrapCallbacks` - Mis à false pour désactiver l'enveloppement automatique des callbacks.

Exemple d'utilisation :

```
echo $this->Js->submit('Save', array('update' => '#content'));
```

Va créé un bouton submit et un événement onclick attaché. L'événement click sera bufferisé par défaut. :

```
echo $this->Js->submit('Save', array('update' => '#content', 'div' => false, 'type' =>
```

Montre comment vous pouvez combiner les options de `FormHelper::submit()` et `Helper Js::request()` à l'utilisation des submits.

link(*$title*, *$url = null*, *$options = array()*)

Créé un élément ancre html qui a un événement clic rattaché. Les options peuvent inclure celle pour `HtmlHelper::link()` et `Helper Js::request()`, `Helper Js::event()`, `$options` est un tableau d'attribut *attributs html* qui sont ajoutés à l'élément ancre généré. Si une option ne fait pas parti des attributs standard de `$htmlAttributes` elle sera passée à `Helper Js::request()` comme une option. Si une Id n'est pas fournie, une valeur aléatoire sera créée pour chacun des liens générés.

Options

–`confirm` - Génère une boite de dialogue de confirmation avant l'envoi de l'événement.

–`id` - utilise une id personnalisée .

–`htmlAttributes` - attributs html non standard supplémentaires. Les attributs standards sont class, id, rel, title, escape, onblur et onfocus.

–`buffer` - Désactive le buffering et retourne une balise script en plus du lien.

Exemple d'utilisation :

```
echo $this->Js->link('Page 2', array('page' => 2), array('update' => '#content'));
```

Va créé un lien pointant vers `/page:2` et mettre à jour #content avec la réponse.

Vous pouvez utiliser les options de `htmlAttributes` pour ajouter des attributs personnalisés. :

```
echo $this->Js->link('Page 2', array('page' => 2), array(
    'update' => '#content',
    'htmlAttributes' => array('other' => 'value')
));

// Créé le html suivant
<a href="/posts/index/page:2" other="value">Page 2</a>
```

serializeForm(*$options = array()*)

Sérialise le formulaire attaché au $selector. Passe `true` pour $isForm si la selection courante est un élément de formulaire. Converti le formulaire ou l'élément de formulaire attaché à la sélection courante dans un objet chaîne/json (dépendant de l'implémentation de la librairie) pour utilisation avec les opérations XHR.

Options

–`isForm` - est ce que la sélection courante est un formulaire ou un input ? (par défaut à false)

–`inline` - est ce que le traitement du rendu sera utilisé dans un autre traitement JS ? (par défaut à false)

En définissant inline == false vous permet de retirer la bordure `;` . Ceci est utile quand vous avez besoin de sérialiser un élément de formulaire comme faisant parti d'une autre opération Javascript ou utilisez la méthode de sérialisation dans un Objet littéral.

redirect(*$url*)

Redirige la page vers `$url` en utilisant `window.location`.

value(*$value*)

Converti une variable native PHP d'un type dans une représentation JSON équivalente. Échappe une valeur de chaîne dans une chaîne compatible JSON. Les caractère UTF-8 seront échappés .

La Pagination Ajax

Bien mieux qu'avec la pagination Ajax de la 1.2, vous pouvez utiliser le Helper JS pour gérer les liens de pagination AJAX au lieu de liens HTML.

Fabriquer les liens Ajax Avant de pouvoir créer les liens ajax vous devez inclure la librairie Javascript qui correspond à l'adaptateur que vous utilisez avec le `Helper JS`. Par défaut le `Helper Js` utilise jQuery. Donc dans votre layout incluez jQuery (ou la librairie que vous utilisez). Assurez vous également d'inclure `RequestHandlerComponent` dans votre behavior. Ajoutez ce qui suit dans votre controller :

```
public $components = array('RequestHandler');
public $helpers = array('Js');
```

Ce qui suit relie la librairie Javascript que vous voulez utiliser. Pour cet exemple nous utiliserons jQuery :

```
echo $this->Html->script('jquery');
```

De même qu'avec la 1.2 vous devez dire au `PaginatorHelper` que vous voulez faire des liens Javascript avancés au lieu des plain HTML. Pour faire cela utilisez `options()` :

```
$this->Paginator->options(array(
    'update' => '#content',
    'evalScripts' => true
));
```

La classe `PaginatorHelper` sait maintenant qu'il faut créer des liens Javascript étendus, et que ces liens devront mettre à jour le contenu `#content` de l'élément. Bien sûr cet élément doit exister, et la plupart du temps vous voulez envelopper le `$content_for_layout` par une div qui correspond à l'id utilisée dans l'option `update`. Vous devez également définir `evalScripts` à true si vous utilisez des adaptateurs Mootools ou Prototype, sans `evalScripts` ces librairies seront incapables de relier les requêtes entrent elles. L'option `indicator` n'est pas supportée par le `Helper JS` et sera ignorée.

Vous venez donc de créer tous les liens demandés pour le fonctionnement de la pagination. Puisque le `Helper Js` bufferise automatiquement tous les contenus de scripts pour réduire les balises `<script>` dans vos codes sources vous **devez** appeler la restitution du buffer. A la fin de votre fichier de vue. Vérifiez l'inclusion de :

```
echo $this->Js->writeBuffer();
```

Si vous oubliez cela vous ne pourrez **pas** enchaîner les liens de paginiation Ajax. Quand vous écrivez le buffer, cela l'efface également , et vous n'avez donc pas à vous inquiéter de doublon de code Javascript.

Ajouter des effets et des transitions Depuis que *indicator*' n'est plus supporté, vous devez ajouter les effets d'indicator vous même. :

```
<!DOCTYPE html>
<html>
    <head>
        <?php echo $this->Html->script('jquery'); ?>
        //plus de trucs ici.
    </head>
    <body>
    <div id="content">
        <?php echo $content_for_layout; ?>
    </div>
    <?php echo $this->Html->image('indicator.gif', array('id' => 'busy-indicator')); ?>
    </body>
</html>
```

Rappelez vous de placer le fichier indicator.gif dans le répertoire app/webroot/img. Vous devriez voir une situation où indicator.gif s'affiche immédiatement au chargement de la page. Vous avez besoin d'insérer cet indicateur `#busy-indicator { display:none; }` dans votre fichier css principal.

Avec le layout ci-dessus, nous avons inclus un indicateur, qui affichera une animation "occupé" que nous aurons à montrer et cacher avec le `Helper Js`. Pour faire cela, nous avons besoin de mettre à jour notre fonction `options()` :

```
$this->Paginator->options(array(
    'update' => '#content',
    'evalScripts' => true,
    'before' => $this->Js->get('#busy-indicator')->effect('fadeIn', array('buffer' => false
    'complete' => $this->Js->get('#busy-indicator')->effect('fadeOut', array('buffer' => fa
));
```

Ceci montrera/cachera l'élément 'indicateur occupé' avant et après que le contenu de la balise `#content` soit mis à jour. Bien que `indicator` ait été enlevé, les nouvelles fonctionnalités du `JsHelper` permettent la création de plus de contrôle et d'effets plus complexes.

NumberHelper

class `NumberHelper` (*View $view*, *array $settings = array()*)

Le helper Number contient des méthodes pratiques qui permettent l'affichage des nombres dans divers formats communs dans vos vues. Ces méthodes contiennent des moyens pour formater les devises, pourcentages, taille des données, le format des nombres avec précisions et aussi de vous donner davantage de souplesse en matière de formatage des nombres. Modifié dans la version 2.1 : `NumberHelper` a été remanié dans une classe `CakeNumber` pour permettre une utilisation plus facile a l'extérieur de la couche `View`. Dans une vue, ces méthodes sont accessibles via la classe *NumberHelper* et vous pouvez l'appeler comme vous pourriez appeler une méthode de helper normale : `$this->Number->method($args);`. Toutes ces fonctions retournent le nombre formaté ; Elles n'affichent pas automatiqement la sortie dans la vue.

`NumberHelper::`**`currency`** (*mixed $number*, *string $currency = 'USD'*, *array $options = array()*)

Paramètres

- **$number** (*float*) – La valeur à convertir.
- **$currency** (*string*) – Le format de monnaie connu à utiliser.
- **$options** (*array*) – Options, voir ci-dessous.

Cette méthode est utilisée pour afficher un nombre dans des formats de monnaie courante (EUR,GBP,USD). L'utilisation dans une vue ressemble à ceci :

```
// Appelé avec NumberHelper
echo $this->Number->currency($number, $currency);

// Appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::currency($number, $currency);
```

Le premier paramètre $number, doit être un nombre à virgule qui représente le montant d'argent que vous désirez. Le second paramètre est utilisé pour choisir un schéma de formatage de monnaie courante :

$currency	1234.56, formaté par le type courant
EUR	€ 1.236,33
GBP	£ 1,236.33
USD	$ 1,236.33

Le troisième paramètre est un tableau d'options pour définir la sortie. Les options suivantes sont disponibles :

Option	Description
before	Le symbole de la monnaie à placer avant les nombres ex : '$'
after	Le symbole de la monnaie à placer après les nombres décimaux ex : 'c'. Définit le boléeen à false pour utiliser aucun symbole décimal ex : 0.35 => $0.35.
zero	Le texte à utiliser pour des valeurs à zéro, peut être une chaîne de caractères ou un nombre. ex : 0, 'Free !'
places	Nombre de décimales à utiliser. ex : 2
thousands	Séparateur des milliers ex : ','
decimals	Symbole de Séparateur des décimales. ex : '.'
negative	Symbole pour les nombres négatifs. Si égal à '()', le nombre sera entouré avec (et)
escape	La sortie doit-elle être échappée de htmlentity ? Par défaut défini à true
wholeSymbol	La chaîne de caractères à utiliser pour les tous nombres. ex : ' dollars'
wholePosition	Soit 'before' soit 'after' pour placer le symbole entier
fractionSymbol	Chaîne de caractères à utiliser pour les nombres en fraction. ex : ' cents'
fractionPosition	Soit 'before' soit 'after' pour placer le symbole de fraction

Si une valeur $currency non reconnue est fournie, elle est préfixée par un nombre formaté en USD. Par exemple :

```
// Appelé avec NumberHelper
echo $this->Number->currency('1234.56', 'FOO');

// Sortie
FOO 1,234.56

// Appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::currency('1234.56', 'FOO');
```

`NumberHelper::defaultCurrency`(*string $currency*)

Paramètres

- **$currency** (*string*) – Défini une monnaie connu pour `CakeNumber::currency()`.

Setter/getter pour la monnaie par défaut. Ceci retire la necessité de toujours passer la monnaie à `CakeNumber::currency()` et change toutes les sorties de monnaie en définissant les autres par défaut. Introduit dans la version 2.3 : Cette méthode a été ajoutée dans 2.3.

`NumberHelper::addFormat`(*string $formatName*, *array $options*)

Paramètres

- **$formatName** (*string*) – Le nom du format à utiliser dans le futur.
- **$options** (*array*) – Le tableau d'options pour ce format. Utilise les mêmes clés `$options` comme `CakeNumber::currency()`.
  - *before* Symbole de monnaie avant le nombre. False pour aucun.
  - *after* Symbole de monnaie après le nombre. False pour aucun.
  - *zero* Le texte à utiliser pour les valeurs à zéro, peut être une chaîne de caractères ou un nombre. ex : 0, 'Free !'
  - *places* Nombre de décimal à utiliser. ex. 2.
  - *thousands* Séparateur des milliers. ex : ',''.
  - *decimals* Symbole de Séparateur des Decimales. ex : '.'.
  - *negative* Symbole pour les nombres négatifs. Si égal à '()', le nombre sera entouré avec (et).
  - *escape* La sortie doit-elle être échappée de htmlentity ? Par défaut à true.
  - *wholeSymbol* Chaîne de caractères à utiliser pour tous les nombres. ex : ' dollars'.
  - *wholePosition* Soit 'before' soit 'after' pour placer le symbole complet.
  - *fractionSymbol* Chaîne de caractères à utiliser pour les nombres à fraction. ex : ' cents'.
  - *fractionPosition* Soit 'before' soit 'after' pour placer le symbole de fraction.

Ajoute le format de monnaie au helper Number. Facilite la réutilisation des formats de monnaie. :

```
// appelé par NumberHelper
$this->Number->addFormat('BRR', array('before' => 'R$ '));

// appelé par CakeNumber
App::uses('CakeNumber', 'Utility');
CakeNumber::addFormat('BRR', array('before' => 'R$ '));
```

Vous pouvez maintenant utiliser *BRR* de manière courte quand vous formattez les montants de monnaie :

```
// appelé par NumberHelper
echo $this->Number->currency($value, 'BRR');

// appelé par CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::currency($value, 'BRR');
```

Les formats ajoutés sont fusionnés avec les formats par défaut suivants :

```
array(
    'wholeSymbol'       => '',
    'wholePosition'     => 'before',
    'fractionSymbol'    => '',
    'fractionPosition'  => 'after',
    'zero'              => 0,
    'places'            => 2,
    'thousands'         => ',',
    'decimals'          => '.',
    'negative'          => '()',
    'escape'            => true
)
```

`NumberHelper::precision` (*mixed $number*, *int $precision = 3*)

Paramètres

- **$number** (*float*) – La valeur à convertir
- **$precision** (*integer*) – Le nombre de décimal à afficher

Cette méthode affiche un nombre avec le montant de précision spécifié (place de la décimal). Elle arrondira afin de maintenir le niveau de précision défini. :

```
// appelé avec NumberHelper
echo $this->Number->precision(456.91873645, 2 );

// Sortie
456.92

// appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::precision(456.91873645, 2 );
```

`NumberHelper::toPercentage` (*mixed $number*, *int $precision = 2*, *array $options = array()*)

Paramètres

- **$number** (*float*) – La valeur à convertir.
- **$precision** (*integer*) – Le nomnbre de décimal à afficher.
- **$options** (*array*) – Options, voir ci-dessous.

Option	Description
multiply	Booléen pour indiquer si la valeur doit être multipliée par 100. Utile pour les pourcentages avec décimal.

Comme precision(), cette méthode formate un nombre selon la précision fournie (où les nombres sont arrondis pour parvenir à ce degré de précision). Cette méthode exprime aussi le nombre en tant que pourcentage et préfixe la sortie avec un signe de pourcent. :

```
// appelé avec NumberHelper. Sortie: 45.69%
echo $this->Number->toPercentage(45.691873645);

// appelé avec CakeNumber. Sortie: 45.69%
App::uses('CakeNumber', 'Utility');
echo CakeNumber::toPercentage(45.691873645);

// Appelé avec multiply. Sortie: 45.69%
echo CakeNumber::toPercentage(0.45691, 2, array(
    'multiply' => true
));
```

Introduit dans la version 2.4 : L'argument `$options` avec l'option `multiply` a été ajouté.

`NumberHelper::fromReadableSize` (*string $size*, *$default*)

Paramètres

- **$size** (*string*) – La valeur formatée lisible par un humain.

Cette méthode enlève le format d'un nombre à partir d'une taille de byte lisible par un humain en un nombre entier de bytes. Introduit dans la version 2.3 : Cette méthode a été ajoutée dans 2.3

`NumberHelper::toReadableSize(string $dataSize)`

Paramètres

– **$data_size** (*string*) – Le nombre de bytes pour le rendre lisible.

Cette méthode formate les tailles de données dans des formes lisibles pour l'homme. Elle fournit une manière raccourcie de convertir les en KB, MB, GB, et TB. La taille est affichée avec un niveau de précision à deux chiffres, selon la taille de données fournie (ex : les tailles supérieurs sont exprimées dans des termes plus larges) :

```
// appelé avec NumberHelper
echo $this->Number->toReadableSize(0); // 0 Bytes
echo $this->Number->toReadableSize(1024); // 1 KB
echo $this->Number->toReadableSize(1321205.76); // 1.26 MB
echo $this->Number->toReadableSize(5368709120); // 5.00 GB

// appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::toReadableSize(0); // 0 Bytes
echo CakeNumber::toReadableSize(1024); // 1 KB
echo CakeNumber::toReadableSize(1321205.76); // 1.26 MB
echo CakeNumber::toReadableSize(5368709120); // 5.00 GB
```

`NumberHelper::format(mixed $number, mixed $options=false)`

Cette méthode vous donne beaucoup plus de contrôle sur le formatage des nombres pour l'utilisation dans vos vues (et est utilisée en tant que méthode principale par la plupart des autres méthodes de NumberHelper). L'utilisation de cette méthode pourrait ressembler à cela :

```
// appelé avec NumberHelper
$this->Number->format($number, $options);

// appelé avec CakeNumber
CakeNumber::format($number, $options);
```

Le paramètre $number est le nombre que vous souhaitez formater pour la sortie. Avec aucun $options fourni, le nombre 1236.334 sortirait comme ceci : 1,236. Notez que la précision par défaut est d'aucun chiffre après la virgule.

Le paramètre $options est là où réside la réelle magie de cette méthode.
–Si vous passez un entier alors celui-ci devient le montant de précision pour la fonction.
–Si vous passez un tableau associatif, vous pouvez utiliser les clés suivantes :
–places (integer) : le montant de précision désiré.
–before (string) : à mettre avant le nombre à sortir.
–escape (boolean) : si vous voulez la valeur avant d'être échappée.
–decimals (string) : utilisé pour délimiter les places des décimales dans le nombre.
–thousands (string) : utilisé pour marquer les milliers, millions, …
Exemple :

```
// appelé avec NumberHelper
echo $this->Number->format('123456.7890', array(
    'places' => 2,
    'before' => '¥ ',
    'escape' => false,
    'decimals' => '.',
```

```
        'thousands' => ','
    ));
    // sortie '¥ 123,456.79'

    // appelé avec CakeNumber
    App::uses('CakeNumber', 'Utility');
    echo CakeNumber::format('123456.7890', array(
        'places' => 2,
        'before' => '¥ ',
        'escape' => false,
        'decimals' => '.',
        'thousands' => ','
    ));
    // sortie '¥ 123,456.79'
```

`NumberHelper::formatDelta` (*mixed $number*, *mixed $options=array()*)

Cette méthode affiche les différences en valeur comme un nombre signé :

```
    // appelé avec NumberHelper
    $this->Number->formatDelta($number, $options);

    // appelé avec CakeNumber
    CakeNumber::formatDelta($number, $options);
```

Le paramètre $number est le nombre que vous planifiez sur le formatage de sortie. Avec aucun $options fourni, le nombre 1236.334 sortirait 1,236. Notez que la valeur de precision par défaut est aucune décimale.

Le paramètre $options prend les mêmes clés que `CakeNumber::format()` lui-même :
–places (integer) : le montant de precision souhaité.
–before (string) : à mettre avant le nombre sorti.
–after (string) : à mettre après le nombre sorti.
–decimals (string) : utilisé pour délimiter les places de la décimal dans un nombre.
–thousands (string) : utilisé pour marquer les places des centaines, millions, ...
Exemple :

```
    // appelé avec NumberHelper
    echo $this->Number->formatDelta('123456.7890', array(
        'places' => 2,
        'decimals' => '.',
        'thousands' => ','
    ));
    // sortie '+123,456.79'

    // appelé avec CakeNumber
    App::uses('CakeNumber', 'Utility');
    echo CakeNumber::formatDelta('123456.7890', array(
        'places' => 2,
        'decimals' => '.',
        'thousands' => ','
    ));
    // sortie '+123,456.79'
```

Introduit dans la version 2.3 : Cette méthode a été ajoutée dans 2.3.

Warning : Depuis 2.4 les symboles sont maintenant en UTF-8. Merci de regarder le guide de migration pour plus de détails si vous lancez une application non-UTF-8.

Paginator

class `PaginatorHelper` (*View $view*, *array $settings = array()*)

Le Helper Paginator est utilisé pour présenter des contrôles de pagination comme les numéros de pages et les liens suivant/précédent. Il travaille en tamdem avec `PaginatorComponent`.

Voir aussi *Pagination* pour des informations sur la façon de créé des jeux de données paginés et faire des requêtes paginées.

Création de liens triés

`PaginatorHelper::sort` (*$key*, *$title = null*, *$options = array()*)

Paramètres

- **$key** (*string*) – Le nom de la clé du jeu d'enregistrement qui doit être triée.
- **$title** (*string*) – Titre du lien. Si $title est null $key sera utilisée pour le titre et sera générée par inflexion.
- **$options** (*array*) – Options pour le tri des liens.

Génère un lien de tri. Définit le nom ou les paramètres de la chaîne de recherche pour le tri et la direction. Les liens par défaut fournirons un tri ascendant. Après le premier clique, les liens générés avec `sort()` gèreront le changement de direction automatiquement. Les liens de tri par défaut ascendant. Si le jeu de résultat est trié en ascendant avec la clé spécifiée le liens retourné triera en 'décroissant'.

Les clés acceptée pour `$options` :
– `escape` Si vous voulez que le contenu soit encoder en html, true par défaut.
– `model` Le model à utiliser, par défaut à PaginatorHelper : :defaultModel().
En considérant que vous paginez des posts, qu'ils sont sur la page un :

```
echo $this->Paginator->sort('user_id');
```

Sortie :

```
<a href="/posts/index/page:1/sort:user_id/direction:asc/">User Id</a>
```

Vous pouvez utiliser le paramètre title pour créer des textes personnalisés pour votre lien :

```
echo $this->Paginator->sort('user_id', 'User account');
```

Sortie :

```
<a href="/posts/index/page:1/sort:user_id/direction:asc/">User account</a>
```

Si vous utilisez du html comme des images dans vos liens rappelez-vous de paramétrer l'échappement :

```
echo $this->Paginator->sort('user_id', '<em>User account</em>', array('escape' => false));
```

Sortie :

```
<a href="/posts/index/page:1/sort:user_id/direction:asc/"><em>User account</em></a>
```

L'option de direction peut être utilisée pour paramétrer la direction par défaut pour un lien. Une fois qu'un lien est activé, il changera automatiquement de direction comme habituellement :

```
echo $this->Paginator->sort('user_id', null, array('direction' => 'desc'));
```

Sortie

```
<a href="/posts/index/page:1/sort:user_id/direction:desc/">User Id</a>
```

`PaginatorHelper::sortDir`(*string $model = null*, *mixed $options = array()*)
récupère la direction courante du tri du jeu d'enregistrement.

`PaginatorHelper::sortKey`(*string $model = null*, *mixed $options = array()*)
récupère la clé courante selon laquelle le jeu d'enregistrement est trié.

Création des liens de page numérotés

`PaginatorHelper::numbers`(*$options = array()*)

Retourne un ensemble de nombres pour le jeu de résultat paginé. Utilise un modulo pour décider combien de nombres à présenter de chaque coté de la page courante. Par défaut 8 liens de chaque coté de la page courante seront créés si cette page existe. Les liens ne seront pas générés pour les pages qui n'existent pas. La page courante n'est pas un lien également.

Les options supportées sont :
- `before` Contenu a insérer avant les nombres.
- `after` Contenu a insérer après les nombres.
- `model` Model pour lequel créer des nombres, par défaut à `PaginatorHelper::defaultModel()`.
- `modulus` combien de nombres à inclure sur chacun des cotés de la page courante, par défaut à 8.
- `separator` Séparateur, par défaut à '' | ''
- `tag` La balise dans laquelle envelopper les liens, par défaut à 'span'.
- `class` Le nom de classe de la balise enveloppante.
- `currentClass` Le nom de classe à utiliser sur le lien courant/actif. Par défaut à *current*.
- `first` Si vous voulez que les premiers liens soit générés, définit à un entier pour définir le nombre de 'premier' liens à générer. Par défaut à false. Si une chaîne est définie un lien pour la première page sera générée avec la valeur comme titre :

```
echo $this->Paginator->numbers(array('first' => 'Première page'));
```

- `last` Si vous voulez que les derniers liens soit générés, définit à un entier pour définir le nombre de 'dernier' liens à générer. Par défaut à false. Suit la même logique que l'option `first`. il y a méthode `last()` à utiliser séparément si vous le voulez.
- `ellipsis` Contenu des suspensions, par défaut à '...'
- `class` Le nom de classe utilisé sur une balise entourante.

– `currentClass` Le nom de classe à utiliser sur le lien courant/actif. Par défaut à *current*.
– La balise `currentTag` à utiliser pour le nombre de page courant, par défaut à null. Cela vous autorise à générer par exemple le Bootstrap Twitter comme les liens avec le nombre de page courant enroulé dans les balises 'a' ou 'span' supplémentaires.

Bien que cette méthode permette beaucoup de customisation pour ses sorties. Elle est aussi prête pour être appelée sans aucun paramètres. :

```
echo $this->Paginator->numbers();
```

En utilisant les options first et last vous pouvez créer des liens pour le début et la fin du jeu de page. Le code suivant pourrait créer un jeu de liens de page qui inclut les liens des deux premiers et deux derniers résultats de pages :

```
echo $this->Paginator->numbers(array('first' => 2, 'last' => 2));
```

Modifié dans la version 2.1 : L'option `currentClass` à été ajoutée dans la version 2.1.Introduit dans la version 2.3 : L'option `currentTag` a été ajoutée dans 2.3.

Création de liens de sauts

En plus de générer des liens qui vont directement sur des numéros de pages spécifiques, vous voudrez souvent des liens qui amènent vers le lien précédent ou suivant, première et dernière pages dans le jeu de données paginées.

`PaginatorHelper::prev`(*$title = '<< Previous'*, *$options = array()*, *$disabledTitle = null*, *$disabledOptions = array()*)

Paramètres

– **$title** (*string*) – Titre du lien.
– **$options** (*mixed*) – Options pour le lien de pagination.
– **$disabledTitle** (*string*) – Titre quand le lien est désactivé, comme quand vous êtes déjà sur la première page, sans page précédente où aller.
– **$disabledOptions** (*mixed*) – Options pour le lien de pagination désactivé.

Génère un lien vers la page précédente dans un jeu d'enregistrements paginés.

`$options` et `$disabledOptions` supportent les clés suivantes :
–`tag` La balise enveloppante que vous voulez utiliser, 'span' par défaut.
–`escape` Si vous voulez que le contenu soit encodé en html, par défaut à true.
–`model` Le model à utiliser, par défaut PaginatorHelper : :defaultModel()
Un simple exemple serait :

```
echo $this->Paginator->prev(' << ' . __('previous'), array(), null, array('class' => '
```

Si vous étiez actuellement sur la secondes pages des posts (articles), vous obtenez le résultat suivant :

```
<span class="prev"><a rel="prev" href="/posts/index/page:1/sort:title/order:desc"><< p
```

Si il n'y avait pas de page précédente vous obtenez :

```
<span class="prev disabled">« previous</span>
```

Vous pouvez changer la balise enveloppante en utilisant l'option `tag`

```
echo $this->Paginator->prev(__('previous'), array('tag' => 'li'));
```

Sortie :

```
<li class="prev"><a rel="prev" href="/posts/index/page:1/sort:title/order:desc">previo
```

Vous pouvez aussi désactiver la balise enroulante :

```
echo $this->Paginator->prev(__('previous'), array('tag' => false));
```

Output :

```
<a class="prev" rel="prev" href="/posts/index/page:1/sort:title/order:desc">previous</
```

Modifié dans la version 2.3 : Pour les méthodes : `PaginatorHelper::prev()` et `PaginatorHelper::next()`, il est maintenant possible de définir l'option `tag` à `false` pour désactiver le wrapper. Les nouvelles options `disabledTag` ont été ajoutées. Si vous laissez vide `$disabledOptions`, le paramètre `$options` sera utilisé. Vous pouvez enregistrer d'autres saisies si les deux groupes d'options sont les mêmes.

`PaginatorHelper::next` (*$title = 'Next >>'*, *$options = array()*, *$disabledTitle = null*, *$disabledOptions = array()*)

Cette méthode est identique a `prev()` avec quelques exceptions. il créé le lien pointant vers la page suivante au lieu de la précédente. elle utilise aussi `next` comme valeur d'attribut rel au lieu de `prev`.

`PaginatorHelper::first` (*$first = '<< first'*, *$options = array()*)

Retourne une première ou un nombre pour les premières pages. Si une chaîne est fournie, alors un lien vers la première page avec le texte fourni sera créé :

```
echo $this->Paginator->first('< first');
```

Ceci créé un simple lien pour la première page. Ne retournera rien si vous êtes sur la première page. Vous pouvez aussi utiliser un nombre entier pour indiquer combien de premier liens paginés vous voulez générer :

```
echo $this->Paginator->first(3);
```

Ceci créera des liens pour les 3 premières pages, une fois la troisième page ou plus atteinte. Avant cela rien ne sera retourné.

Les paramètres d'option acceptent ce qui suit :
–`tag` La balise tag enveloppante que vous voulez utiliser, par défaut à 'span'.
–`after` Contenu à insérer après le lien/tag.
–`model` Le model à utiliser par défaut PaginatorHelper : :defaultModel().
–`separator` Contenu entre les liens générés, par défaut à ' | '.
–`ellipsis` Contenu pour les suspensions, par défaut à '...'.

`PaginatorHelper::last` (*$last = 'last >>'*, *$options = array()*)

Cette méthode fonctionne très bien comme la méthode `first()`. Elle a quelques différences cependant. Elle ne générera pas de lien si vous êtes sur la dernière page avec la valeur chaîne `$last`. Pour une valeur entière de `$last` aucun lien ne sera généré une fois que l'utilisateur sera dans la zone des dernières pages.

`PaginatorHelper::current` (*string $model = null*)

récupère la page actuelle pour le jeu d'enregistrement du model donné :

```
    // Ou l'url est: http://example.com/comments/view/page:3
    echo $this->Paginator->current('Comment');
    // la sortie est 3
```

`PaginatorHelper::hasNext(string $model = null)`
Retourne true si le résultat fourni n'est pas sur la dernière page.

`PaginatorHelper::hasPrev(string $model = null)`
Retourne true si le résultat fourni n'est pas sur la première page.

`PaginatorHelper::hasPage(string $model = null, integer $page = 1)`
Retourne true si l'ensemble de résultats fourni a le numéro de page fourni par `$page`.

Création d'un compteur de page

`PaginatorHelper::counter($options = array())`

Retourne une chaîne compteur pour le jeu de résultat paginé. En Utilisant une chaîne formatée fournie et un nombre d'options vous pouvez créer des indicateurs et des éléments spécifiques de l'application indiquant ou l'utilisateur se trouve dans l'ensemble de données paginées.

Il y a un certain nombre d'options supportées pour `counter()`. celles supportées sont :

- `format` Format du compteur. Les formats supportés sont 'range', 'pages' et custom. Par défaut à pages qui pourrait ressortir comme '1 of 10'. Dans le mode custom la chaîne fournie est analysée (parsée) et les jetons sont remplacées par des valeurs réelles. Les jetons autorisés sont :
  - `{:page}` - la page courante affichée.
  - `{:pages}` - le nombre total de pages.
  - `{:current}` - le nombre actuel d'enregistrements affichés.
  - `{:count}` - le nombre total d'enregistrements dans le jeu de résultat.
  - `{:start}` - le nombre de premier enregistrement affichés.
  - `{:end}` - le nombre de dernier enregistrements affichés.
  - `{:model}` - La forme plurielle du nom de model. Si votre model était 'RecettePage', `{:model}` devrait être 'recipe pages'. cette option a été ajoutée dans la 2.0.

  Vous pouvez aussi fournir simplement une chaîne à la méthode counter en utilisant les jetons autorisés. Par exemple :

```
echo $this->Paginator->counter(
    'Page {:page} of {:pages}, showing {:current} records out of
     {:count} total, starting on record {:start}, ending on {:end}'
);
```

  En définissant 'format' à 'range' donnerait en sortie '1 - 3 of 13' :

```
echo $this->Paginator->counter(array(
    'format' => 'range'
));
```

- `separator` Le séparateur entre la page actuelle et le nombre de pages. Par défaut à ' of '. Ceci est utilisé en conjonction avec 'format' ='pages' qui la valeur par défaut de 'format' :

```
echo $this->Paginator->counter(array(
    'separator' => ' sur un total de '
));
```

– `model` Le nom du model en cours de pagination, par défaut à `PaginatorHelper::defaultModel()`. Ceci est utilisé en conjonction avec la chaîne personnalisée de l'option 'format'.

Modification des options que le Helper Paginator utilise

`PaginatorHelper::options`(*$options = array()*)

Paramètres

– **$options** (*mixed*) – Options par défaut pour les liens de pagination. Si une chaîne est fournie - elle est utilisée comme id de l'élément DOM à actualiser.

Définit toutes les options pour le Helper Paginator Helper. Les options supportées sont :

– `url` L'url de l'action de pagination. url comporte quelques sous options telles que :
  – `sort` La clé qui décrit la façon de trier les enregistrements.
  – `direction` La direction du tri. Par défaut à 'ASC'.
  – `page` Le numéro de page à afficher.

  Les options mentionnées ci-dessus peuvent être utilisées pour forcer des pages/directions particulières. Vous pouvez aussi ajouter des contenu d'url supplémentaires dans toutes les urls générées dans le helper :

```
$this->Paginator->options(array(
    'url' => array(
        'sort' => 'email', 'direction' => 'desc', 'page' => 6,
        'lang' => 'en'
    )
));
```

  Ce qui se trouve ci-dessus ajoutera `en` comme paramètre de route pour chacun des liens que le helper va générer. Il créera également des liens avec des tris, direction et valeurs de page spécifiques. Par défaut PaginatorHelper fusionnera cela dans tous les paramètres passés et nommés. Ainsi vous n'aurez pas à le faire dans chacun des fichiers de vue.

– `escape` Définit si le champ titre des liens doit être échappé HTML. Par défaut à true.
– `update` Le selecteur CSS de l'élément à actualiser avec le résultat de l'appel de pagination AJAX. Si cela n'est pas spécifié, des liens réguliers seront créés :

```
$this->Paginator->options('update' => '#content');
```

  Ceci est utile lors de l'utilisation de la pagination AJAX *La Pagination Ajax*. Gardez à l'esprit que la valeur actualisée peut être un selecteur CSS valide, mais il est souvent plus simple d'utiliser un selecteur id.

– `model` Le nom du model en cours de pagination, par défaut à `PaginatorHelper::defaultModel()`.

Utilisation de paramètres GET pour la pagination Normalement la Pagination dans CakePHP utilise *Paramètres nommés*. Il y a des fois ou vous souhaiterez utilisez des paramètres GET à la place. Alors que la principale option de configuration pour cette fonctionnalité est dans `PaginatorComponent`, vous avez

des contrôles supplémentaires dans les vues. Vous pouvez utiliser *options()*' pour indiquer que vous voulez la conversion d'autres paramètres nommés :

```
$this->Paginator->options(array('convertKeys' => array('your', 'keys', 'here')));
```

Configurer le Helper Paginator pour utiliser le Helper Javascript Par défaut le `Helper Paginator` utilise `JsHelper` pour effectuer les fonctionnalités Ajax. Toutefois, si vous ne voulez pas cela et que vous voulez utiliser un Helper personnalisé pour les liens Ajax, vous pouvez le faire en changeant le tableau `$helpers` dans votre controller. Après avoir lancé `paginate()` faîtes ce qui suit :

```
// Dans l'action de votre controller.
$this->set('posts', $this->paginate());
$this->helpers['Paginator'] = array('ajax' => 'CustomJs');
```

Changera le `Helper Paginator` pour utiliser `CustomJs` pour les opérations Ajax. Vous pourriez aussi définir la clé Ajax pour être un Helper, tant que la classe implémente la méthode `link()` qui se comporte comme `HtmlHelper::link()`.

La Pagination dans les Vues

C'est à vous de décider comment afficher les enregistrements à l'utilisateur, mais la plupart des fois, ce sera fait à l'intérieur des tables HTML. L'exemple ci-dessous suppose une présentation tabulaire, mais le Helper Paginator disponible dans les vues N'a pas toujours besoin d'être limité en tant que tel.

Voir les détails sur PaginatorHelper (http ://api20.cakephp.org/class/paginator-helper) dans l' API. Comme mentionné précédemment, le Helper Paginator offre également des fonctionnalités de tri qui peuvent être facilement intégrés dans vos en-têtes de colonne de table :

```
// app/View/Posts/index.ctp
<table>
    <tr>
        <th><?php echo $this->Paginator->sort('id', 'ID'); ?></th>
        <th><?php echo $this->Paginator->sort('title', 'Title'); ?></th>
    </tr>
       <?php foreach ($data as $recipe): ?>
    <tr>
        <td><?php echo $recipe['Recipe']['id']; ?> </td>
        <td><?php echo h($recipe['Recipe']['title']); ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

Les liens en retour de la méthode `sort()` du `PaginatorHelper` permettent au utilisateurs de cliquer sur les entêtes de table pour faire basculer l'ordre de tri des données d'un champ donné.

Il est aussi possible de trier une colonne basée sur des associations :

```
<table>
    <tr>
        <th><?php echo $this->Paginator->sort('titre', 'Titre'); ?></th>
```

```
        <th><?php echo $this->Paginator->sort('Auteur.nom', 'Auteur'); ?></th>
    </tr>
       <?php foreach ($data as $recette): ?>
    <tr>
        <td><?php echo h($recette['Recette']['titre']); ?> </td>
        <td><?php echo h($recette['Auteur']['nom']); ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

L'ingrédient final pour l'affichage de la pagination dans les vues est l'addition de pages de navigation, aussi fournies par le Helper de Pagination :

```
// Montre les numéros de page
echo $this->Paginator->numbers();

// Montre les liens précédent et suivant
echo $this->Paginator->prev('« Previous', null, null, array('class' => 'disabled'));
echo $this->Paginator->next('Next »', null, null, array('class' => 'disabled'));

// affiche X et Y, ou X est la page courante et Y est le nombre de pages
echo $this->Paginator->counter();
```

Le texte de sortie de la méthode counter() peut également être personnalisé en utilisant des marqueurs spéciaux :

```
echo $this->Paginator->counter(array(
    'format' => 'Page {:page} of {:pages}, showing {:current} records out of
             {:count} total, starting on record {:start}, ending on {:end}'
));
```

D'autres Méthodes

`PaginatorHelper::link`(*$title*, *$url = array()*, *$options = array()*)

Paramètres

- **$title** (*string*) – Titre du lien.
- **$url** (*mixed*) – Url de l'action. Voir Router : :url().
- **$options** (*array*) – Options pour le lien. Voir options() pour la liste des clés.

Les clés acceptées pour `$options` :
–**update - L' Id de l'élément DOM que vous souhaitez actualiser.** Créé des liens près pour Ajax.
–**escape Si vous voulez que le contenu soit encodé comme une** entité html, par défaut à true.
–**model Le model à utiliser, par défaut à** PaginatorHelper : :defaultModel().
Créé un lien ordinaire ou Ajax avec des paramètres de pagination :

```
echo $this->Paginator->link('Sort by title on page 5',
        array('sort' => 'title', 'page' => 5, 'direction' => 'desc'));
```

Si créé dans la vue de `/posts/index`, cela créerait un lien pointant vers '/posts/index/page :5/sort :title/direction :desc'.

`PaginatorHelper::url`(*$options = array()*, *$asArray = false*, *$model = null*)

Paramètres

- **$options** (*array*) – Tableau d'options Pagination/URL. Comme utilisé dans les méthodes `options()` ou `link()`.
- **$asArray** (*boolean*) – Retourne l'url comme dans un tableau, ou une chaîne URL. Par défaut à false.
- **$model** (*string*) – Le model sur lequel paginer.

Par défaut retourne une chaîne URL complètement paginée à utiliser dans des contextes non-standard (ex. JavaScript). :

```
echo $this->Paginator->url(array('sort' => 'titre'), true);
```

`PaginatorHelper::defaultModel()`

Retourne le model par défaut du jeu de pagination ou null si la pagination n'est pas initialisée.

`PaginatorHelper::params`(*string $model = null*)

Retourne les paramètres courants de la pagination du jeu de résultat d'un model donné :

```
debug($this->Paginator->params());
/*
Array
(
    [page] => 2
    [current] => 2
    [count] => 43
    [prevPage] => 1
    [nextPage] => 3
    [pageCount] => 3
    [order] =>
    [limit] => 20
    [options] => Array
        (
            [page] => 2
            [conditions] => Array
                (
                )
        )
    [paramType] => named
)
*/
```

`PaginatorHelper::param`(*string $key*, *string $model = null*)

Récupère le paramètre de pagination spécifique à partir de l'ensemble de résultats pour le model donné :

```
debug($this->Paginator->param('count'));
/*
(int)43
*/
```

Introduit dans la version 2.4 : La méthode `param()` a été ajoutée dans 2.4.

RSS

class `RssHelper` (*View $view*, *array $settings = array()*)

Le Helper RSS permet de générer facilement des XML pour les flux RSS.

Créer un flux RSS avec RssHelper

Cet exemple suppose que vous ayez un Controller Posts et un Model Post déjà créés et que vous vouliez créer une vue alternative pour les flux RSS.

Créer une version xml/rss de posts/index est vraiment simple avec CakePHP. Après quelques étapes faciles, vous pouvez tout simplement ajouter l'extension .rss demandée à `posts/index` pour en faire votre URL `posts/index.rss`. Avant d'aller plus loin en essayant d'initialiser et de lancer notre service Web, nous avons besoin de faire un certain nombre de choses. Premièrement, le parsing d'extensions doit être activé dans `app/config/routes.php` :

```
Router::parseExtensions('rss');
```

Dans l'appel ci-dessus, nous avons activé l'extension .rss. Quand vous utilisez `Router::parseExtensions()`, vous pouvez passer autant d'arguments ou d'extensions que vous le souhaitez. Cela activera le content-type de chaque extension utilisée dans votre application. Maintenant, quand l'adresse `posts/index.rss` est demandée, vous obtiendrez une version XML de votre `posts/index`. Cependant, nous avons d'abord besoin d'éditer le controller pour y ajouter le code "rss-spécifique".

Code du Controller C'est une bonne idée d'ajouter RequestHandler au tableau $components de votre controller Posts. Cela permettra à beaucoup d'automagie de se produire :

```
public $components = array('RequestHandler');
```

Notre vue utilise aussi `TextHelper` pour le formatage, ainsi il doit aussi être ajouté au controller :

```
public $helpers = array('Text');
```

Avant que nous puissions faire une version RSS de notre posts/index, nous avons besoin de mettre certaines choses en ordre. Il pourrait être tentant de mettre le canal des métadonnées dans l'action du controller et de le passer à votre vue en utilisant la méthode `Controller::set()`, mais ceci est inapproprié. Cette information pourrait également aller dans la vue. Cela arrivera sans doute plus tard, mais pour l'instant, si vous avez un ensemble de logique différent entre les données utilisées pour créer le flux RSS et les données pour la page HTML, vous pouvez utiliser la méthode `RequestHandler::isRss()`, sinon votre controller pourrait rester le même :

```
// Modifie l'action du Controller Posts correspondant à
// l'action qui délivre le flux rss, laquelle est
// l'action index dans notre exemple

public function index() {
    if ($this->RequestHandler->isRss() ) {
```

```
        $posts = $this->Post->find('all', array('limit' => 20, 'order' => 'Post.created DES
        return $this->set(compact('posts'));
    }

    // ceci n'est pas une requête RSS
    // donc on retourne les données utilisées par l'interface du site web
    $this->paginate['Post'] = array('order' => 'Post.created DESC', 'limit' => 10);

    $posts = $this->paginate();
    $this->set(compact('posts'));
}
```

Maintenant que toutes ces variables de Vue sont définies, nous avons besoin de créer un layout rss.

Layout Un layout Rss est très simple, mettez les contenus suivants dans `app/View/Layouts/rss/default.ctp`:

```
if (!isset($documentData)) {
    $documentData = array();
}
if (!isset($channelData)) {
    $channelData = array();
}
if (!isset($channelData['title'])) {
    $channelData['title'] = $title_for_layout;
}
$channel = $this->Rss->channel(array(), $channelData, $content_for_layout);
echo $this->Rss->document($documentData,$channel);
```

Il ne ressemble pas à plus mais grâce à la puissance du `RssHelper` il fait beaucoup pour améliorer le visuel pour nous. Nous n'avons pas défini `$documentData` ou `$channelData` dans le controller, cependant dans CakePHP vos vues peuvent retourner les variables au layout. Ce qui est l'endroit où notre tableau `$channelData` va venir définir toutes les données meta pour notre flux.

Ensuite il y a le fichier de vue pour mes posts/index. Un peu comme le fichier de layout que nous avons crée, nous avons besoin de créer un répertoire `View/Posts/rss/` et un nouveau `index.ctp` à l'intérieur de ce répertoire. Les contenus du fichier sont ci-dessous.

View Notre vue, localisée dans `app/View/Posts/rss/index.ctp`, commence par définir les variables `$documentData` et `$channelData` pour le layout, celles-ci contiennent toutes les metadonnées pour notre flux RSS. C'est fait en utilisant la méthode `View::set()` ' qui est analogue à la méthode Controller : :set(). Ici nous passons les canaux de données en retour au layout :

```
$this->set('channelData', array(
    'title' => __("Most Recent Posts"),
    'link' => $this->Html->url('/', true),
    'description' => __("Most recent posts."),
    'language' => 'en-us'));
```

La seconde partie de la vue génère les éléments pour les enregistrements actuels du flux. Ceci est accompli en bouclant sur les données qui ont été passées à la vue ($items) et en utilisant la méthode

`RssHelper::item()`. L'autre méthode que vous pouvez utiliser `RssHelper::items()` qui prend un callback et un tableau des items pour le flux. (La méthode que j'ai vu utilisée pour le callback a toujours été appelée `transformRss()`. Il y a un problème pour cette méthode, qui est qu'elle n'utilise aucune des classes de helper pour préparer vos données à l'intérieur de la méthode de callback parce que la portée à l'intérieur de la méthode n'inclut pas tout ce qui n'est pas passé à l'intérieur, ainsi ne donne pas accès au TimeHelper ou à tout autre helper dont vous auriez besoin. `RssHelper::item()` transforme le tableau associatif en un élément pour chaque pair de valeur de clé.

Note : Vous devrez modifier la variable $postLink comme il se doit pour votre application.

```
// Vous devez importer Sanitize
App::uses('Sanitize', 'Utility');

foreach ($posts as $post) {
    $postTime = strtotime($post['Post']['created']);

    $postLink = array(
        'controller' => 'posts',
        'action' => 'view',
        'year' => date('Y', $postTime),
        'month' => date('m', $postTime),
        'day' => date('d', $postTime),
        $post['Post']['slug']
    );

    // C\'est la partie où nous nettoyons le corps du teste pour la sortie en description
    // de l\item de rss, cela nécessite d'avoir seulement le text pour s'assurer le validat
    $bodyText = preg_replace('=\(.*?\)=is', '', $post['Post']['body']);
    $bodyText = $this->Text->stripLinks($bodyText);
    $bodyText = Sanitize::stripAll($bodyText);
    $bodyText = $this->Text->truncate($bodyText, 400, array(
        'ending' => '...',
        'exact'  => true,
        'html'   => true,
    ));

    echo  $this->Rss->item(array(), array(
        'title' => $post['Post']['title'],
        'link' => $postLink,
        'guid' => array('url' => $postLink, 'isPermaLink' => 'true'),
        'description' => $bodyText,
        'pubDate' => $post['Post']['created']
    ));
}
```

Vous pouvez voir ci-dessus que nous pouvons utiliser la boucle pour préparer les données devant être transformées en elements XML. Il est important de filtrer tout texte de caractères non brute en-dehors de la description, spécialement si vous utilisez un éditeur de texte riche pour le corps de votre blog. Dans le code ci-dessus nous utilisons la méthode `TextHelper::stripLinks()` et quelques méthodes de la classe Sanitize, mais nous recommandons d'écrire un helper propre compréhensible pour réellement capturer le texte propre. Une fois que nous avons défini les données pour le feed, nous pouvons ensuite utiliser la méth-

ode `RssHelper::item()` pour créer le XML dans le format RSS. Une fois que vous avez toutes ces configurations, vous pouvez tester votre feed RSS en allant à votre `/posts/index.rss` et que vous verrez votre nouveau feed. Il est toujours important que vous validiez votre feed RSS avant de le mettre en live. Ceci peut être fait en visitant les sites qui valident le XML comme Le Validateur de Feed ou le site de w3c à http ://validator.w3.org/feed/.

Note : Vous aurez besoin de définir la valeur de 'debug' dans votre configuration du coeur à 1 ou à 0 pour obtenir un flux valide, à cause des différentes informations de debug ajoutées automatiquement sous des paramètres de debug plus haut qui cassent la syntaxe XML ou les règles de validation du flux.

API de Rss Helper

property `RssHelper::$action`
Action courante

property `RssHelper::$base`
Base URL

property `RssHelper::$data`
donnée du model POSTée

property `RssHelper::$field`
Nom du champ courant

property `RssHelper::$helpers`
Helpers utilisés par le Helper RSS

property `RssHelper::$here`
URL de l'action courante

property `RssHelper::$model`
Nom du model courant

property `RssHelper::$params`
Paramètre tableau

property `RssHelper::$version`
Version de spec par défaut de la génération de RSS.

`RssHelper::channel`(*array $attrib = array ()*, *array $elements = array ()*, *mixed $content = null*)

Type retourné string

Retourne un élément RSS `<channel />`.

`RssHelper::document`(*array $attrib = array ()*, *string $content = null*)

Type retourné string

Retourne un document RSS entouré de tags `<rss />`.

`RssHelper::elem`(*string $name*, *array $attrib = array ()*, *mixed $content = null*, *boolean $endTag = true*)

Type retourné string

Génére un élément XML.

`RssHelper::item`(*array $att = array ()*, *array $elements = array ()*)

Type retourné string

Convertit un tableau en un élément `<item />` et ses contenus.

`RssHelper::items`(*array $items*, *mixed $callback = null*)

Type retourné string

Transforme un tableau de données en utilisant un callback optionnel, et le map pour un ensemble de tags `<item />`.

`RssHelper::time`(*mixed $time*)

Type retourné string

Convertit un time de tout format en time de RSS. Regardez `TimeHelper::toRSS()`.

SessionHelper

class `SessionHelper`(*View $view*, *array $settings = array()*)

Équivalent du Component Session, le Helper Session offre la majorité des fonctionnalités du component et les rend disponible dans votre vue. Le Helper session est automatiquement ajouté à la vue, il n'est pas nécessaire de l'ajouter à la variable tableau `$helpers` dans votre controller.

La grande différence entre le Component Session et le Helper Session est que ce dernier *ne peut pas* écrire dans la session.

Comme pour le Component Session, les données sont écrites et lues en utilisant des structures de tableaux avec des *dot notation*, comme ci-dessous :

```
array('User' =>
    array('username' => 'super@example.com')
);
```

Étant donné ce tableau, le nœud sera accessible par `User.username`, le point indiquant le tableau imbriqué. Cette notation est utilisée pour toutes les méthodes du helper Session où une variable `$key` est utilisée.

`SessionHelper::read`(*string $key*)

Type retourné mixed

Lire à partir de la Session. Retourne une chaîne de caractère ou un tableau dépendant des contenus de la session.

`SessionHelper::check`(*string $key*)

Type retourné boolean

Vérifie si une clé est dans la Session. Retourne un boléen sur l'existence d'un clé.

`SessionHelper::error`()

Type retourné string

Retourne la dernière erreur encourue dans une session.

`SessionHelper::valid()`

Type retourné boolean

Utilisée pour vérifier si une session est valide dans une vue.

Affichage de notifications ou de messages flash

`SessionHelper::flash(string $key = 'flash', array $params = array())`

Type retourné string

Comme expliqué dans *Création de messages de notification* vous pouvez créer des notifications uniques pour le le feedback. Après avoir créé les messages avec `SessionComponent::setFlash()`, vous voudrez les afficher. Une fois que le message est affiché, il sera retiré et ne s'affichera plus :

```
echo $this->Session->flash();
```

Ce qui est au-dessus sortira un message simple, avec le html suivant :

```
<div id="flashMessage" class="message">
    Vos trucs on été sauvegardés.
</div>
```

Comme pour la méthode du component, vous pouvez définir des propriétés supplémentaires et personnaliser quel élément est utilisé. Dans le controller, vous pouvez avoir du code comme :

```
// dans un controller
$this->Session->setFlash('Le user n'a pu être supprimé.');
```

Quand le message sort, vous pouvez choisir l'élément utilisé pour afficher ce message :

```
// dans un layout.
echo $this->Session->flash('flash', array('element' => 'failure'));
```

Ceci utilise `View/Elements/failure.ctp` pour rendre le message. Le message texte sera disponible dans une variable `$message` dans l'élément.

A l'intérieur du fichier élément d'échec, il y aura quelque chose comme ceci :

```
<div class="flash flash-failure">
    <?php echo $message ?>
</div>
```

Vous pouvez aussi passer des paramètres supplémentaires dans la méthode `flash()`, ce qui vous permet de générer des messages personnalisés :

```
// Dans le controller
$this->Session->setFlash('Thanks for your payment %s');

// Dans le layout.
echo $this->Session->flash('flash', array(
    'params' => array('name' => $user['User']['name'])
    'element' => 'payment'
```

```
    ));

    // View/Elements/payment.ctp
    <div class="flash payment">
        <?php printf($message, h($name)); ?>
    </div>
```

TextHelper

class TextHelper (*View $view*, *array $settings = array()*)

TextHelper possède des méthodes pour rendre le texte plus utilisable et sympa dans vos vues. Il aide à activer les liens, à formater les urls, à créer des extraits de texte autour des mots ou des phrases choisies, mettant en évidence des mots clés dans des blocs de texte et tronquer élegamment de longues étendues de texte. Modifié dans la version 2.1 : Plusieurs des méthodes de `TextHelper` ont été déplacées dans la classe `String` pour permettre une utilisation plus facile de la couche `View`. Dans une vue, ces méthodes sont accessibles avec la classe *TextHelper* et vous pouvez l'appeler comme vous appelleriez une méthode normale de helper : `$this->Text->method($args);`.

`TextHelper::`**autoLinkEmails** (*string $text*, *array $options=array()*)

Paramètres

- **$text** (*string*) – Le texte à convertir.
- **$options** (*array*) – Un tableau de *html attributes* pour générer les liens.

Ajoute les liens aux adresses email bien formées dans $text, selon toute les options définies dans `$htmlOptions` (regardez `HtmlHelper::link()`). :

```
$my_text = 'Pour plus d'informations sur nos pâtes et desserts fameux, contactez info
$linked_text = $this->Text->autoLinkEmails($my_text);
```

Sortie :

```
Pour plus d' informations sur nos pâtes et desserts fameux,
contactez <a href="mailto:info@example.com">info@example.com</a>
```

Modifié dans la version 2.1 : Dans 2.1, cette méthode echappe automatiquement ces inputs. Utilisez l'option `escape` pour la désactiver si nécéssaire.

`TextHelper::`**autoLinkUrls** (*string $text*, *array $htmlOptions=array()*)

Paramètres

- **$text** (*string*) – Le texte à convertir.
- **$htmlOptions** (*array*) – Un tableau de *html attributes* pour la génération de liens.

De même que dans `autoLinkEmails()`, seule cette méthode cherche les chaînes de caractère qui commence par https, http, ftp, ou nntp et les liens de manière appropriée. Modifié dans la version 2.1 : Dans 2.1, cette méthode échappe automatiquement son input. Utilisez l'option `escape` pour la désactiver si nécessaire.

`TextHelper::`**autoLink** (*string $text*, *array $htmlOptions=array()*)

Paramètres

- **$text** (*string*) – Le texte à lier automatiquement.

– **$htmlOptions** (*array*) – Un tableau de *html attributes* pour générer les liens.

Execute la fonctionnalité dans les deux `autoLinkUrls()` et `autoLinkEmails()` sur le `$text` fourni. Tous les URLs et emails sont liés de manière appropriée donnée par `$htmlOptions` fourni. Modifié dans la version 2.1 : Dans 2.1, cette méthode échappe automatiquement son input. Utilisez l'option `escape` pour la désactiver si nécéssaire.

`TextHelper::autoParagraph` (*string $text*)

Paramètres

– **$text** (*string*) – Le texte à convertir.

Ajoute <p> autour du texte où la double ligne retourne et <br> où une simple ligne retourne, sont trouvés. :

```
$myText = 'For more information
regarding our world-famous pastries and desserts.

contact info@example.com';
$formattedText = $this->Text->autoParagraph($myText);
```

Output :

```
<p>Pour plus d\' information<br />
selon nos c é l è bres p â tes et desserts.<p>
<p>contact info@example.com</p>
```

Introduit dans la version 2.4.

`TextHelper::highlight` (*string $haystack*, *string $needle*, *array $options = array()*)

Paramètres

– **$haystack** (*string*) – La chaîne de caractères à rechercher.
– **$needle** (*string*) – La chaîne à trouver.
– **$options** (*array*) – Un tableau d'options, voir ci-dessous.

Mettre en avant `$needle` dans `$haystack` en utilisant la chaîne spécifique `$options['format']` ou une chaîne par défaut.

Options :
–'format' - chaîne la partie de html avec laquelle la phrase sera mise en excergue.
–'html' - bool Si true, va ignorer tous les tags HTML, s'assurant que seul le bon texte est mise en avant.

Exemple :

```
// appelé avec TextHelper
echo $this->Text->highlight($last_sentence, 'using', array('format' => '<span class="h

// appelé avec String
App::uses('String', 'Utility');
echo String::highlight($last_sentence, 'using', array('format' => '<span class="highli
```

Sortie :

```
Highlights $needle in $haystack <span class="highlight">using</span>
the $options['format'] string specified  or a default string.
```

`TextHelper::stripLinks`(*$text*)

Enlève le `$text` fourni de tout lien HTML.

`TextHelper::truncate`(*string $text*, *int $length=100*, *array $options*)

Paramètres

- **$text** (*string*) – Le texte à tronquer.
- **$length** (*int*) – La longueur de trim.
- **$options** (*array*) – Un tableau d'options à utiliser.

Coupe une chaîne avec `$length` et ajoute un suffixe avec `'ending'` si le texte est plus long que `$length`. Si `'exact'` est passé à `false`, le truchement va se faire après le mot de fin suivant. Si `'html'` est passé à `true`, les balises html seront respectés et ne seront pas coupés.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et a les clés suivantes possibles par défaut, celles-ci étant toutes optionnelles :

```
array(
    'ellipsis' => '...',
    'exact' => true,
    'html' => false
)
```

Exemple :

```
// appelé avec TextHelper
echo $this->Text->truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    array(
        'ellipsis' => '...',
        'exact' => false
    )
);

// appelé avec String
App::uses('String', 'Utility');
echo String::truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    array(
        'ellipsis' => '...',
        'exact' => false
    )
);
```

Sortie :

```
The killer crept...
```

Modifié dans la version 2.3 : `ending` a été remplacé par `ellipsis`. `ending` est toujours utilisé dans 2.2.1.

`TextHelper::tail`(*string $text*, *int $length=100*, *array $options*)

Paramètres

- **$text** (*string*) – The text à tronquer.

- **$length** (*int*) – La longueur de trim.
- **$options** (*array*) – Un tableau d'options à utiliser.

Coupe une chaîne à la longueur `$length` et ajoute un préfixe avec `'ellipsis'` si le texte est plus long que `$length`. Si `'exact'` est passé à `false`, le truchement se fera avant le prochain mot de fin.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et a les clés possibles suivantes par défaut, toutes sont optionnelles :

```
array(
    'ellipsis' => '...',
    'exact' => true
)
```

Introduit dans la version 2.3. Exemple :

```
// appelé avec TextHelper
echo $this->Text->tail(
    'I packed my bag and in it I put a PSP, a PS3, a TV, a C# program that can divide
    70,
    array(
        'ellipsis' => '...',
        'exact' => false
    )
);

// appelé avec String
App::uses('String', 'Utility');
echo String::tail(
    'I packed my bag and in it I put a PSP, a PS3, a TV, a C# program that can divide
    70,
    array(
        'ellipsis' => '...',
        'exact' => false
    )
);
```

Sortie :

```
...a TV, a C# program that can divide by zero, death metal t-shirts
```

`TextHelper::`**`excerpt`**(*string $haystack*, *string $needle*, *integer $radius=100*, *string $ending="..."*)

Paramètres

- **$haystack** (*string*) – La chaîne à chercher.
- **$needle** (*string*) – La chaîne to excerpt around.
- **$radius** (*int*) – Le nombre de caractères de chaque côté de $needle que vous souhaitez inclure.
- **$ending** (*string*) – Le Texte à ajouter/préfixer au début ou à la fin du résultat.

Extrait un excerpt de `$haystack` surrounding the `$needle` avec un nombre de caractères de chaque côté determiné par `$radius`, and prefix/suffix with `$ending`. Cette méthode est spécialement pratique pour les résultats recherchés. La chaîne requêtée ou les mots clés peuvent être montrés dans le document résultant. :

```
// appelé avec TextHelper
echo $this->Text->excerpt($last_paragraph, 'method', 50, '...');

// appelé avec String
App::uses('String', 'Utility');
echo String::excerpt($last_paragraph, 'method', 50, '...');
```

Sortie :

```
... par $radius, et prefix/suffix avec $ending. Cette méthode est
spécialement pratique pour les résultats de recherche. La requête...
```

`TextHelper::toList` (*array $list*, *$and='and'*)

Paramètres

- **$list** (*array*) – Tableau d'éléments à combiner dans une list sentence.
- **$and** (*string*) – Le mot utilisé pour le dernier join.

Crée une liste séparée avec des virgules, où les deux derniers items sont joins avec 'and'. :

```
// appelé avec TextHelper
echo $this->Text->toList($colors);

// appelé avec String
App::uses('String', 'Utility');
echo String::toList($colors);
```

Sortie :

```
red, orange, yellow, green, blue, indigo et violet
```

TimeHelper

class `TimeHelper` (*View $view*, *array $settings = array()*)

Le Helper Time vous permet, comme il l'indique de gagner du temps. Il permet le traitement rapide des informations se rapportant au temps. Le Helper Time a deux tâches principales qu'il peut accomplir :

1. Il peut formater les chaînes de temps.
2. Il peut tester le temps (mais ne peut pas le courber, désolé).

Modifié dans la version 2.1 : `TimeHelper` a été reconstruit dans la classe `CakeTime` pour faciliter l'utilisation en-dehors de la couche `View`. Dans une vue, ces méthodes sont accessibles via la classe *TimeHelper* et vous pouvez l'appeler comme vous appelleriez une méthode normale de helper : `$this->Time->method($args);`.

Utiliser le Helper

Une utilisation courante de Time Helper est de compenser la date et le time pour correspondre au time zone de l'utilisateur. Utilisons un exemple de forum. Votre forum a plusieurs utilisateurs qui peuvent poster des messages depuis n'importe quelle partie du monde. Une façon facile de gérer le

temps est de sauvegarder toutes les dates et les times à GMT+0 or UTC. Décommenter la ligne `date_default_timezone_set('UTC');` dans `app/Config/core.php` pour s'assurer que le time zone de votre aplication est défini à GMT+0.

Ensuite, ajoutez un time zone à votre table users et faîtes les modifications nécessaires pour permettre à vos utilisateurs de définir leur time zone. Maintenant que nous connaissons le time zone de l'utilisateur connecté, nous pouvons corriger la date et le temps de nos posts en utilisant le Helper Time :

```
echo $this->Time->format('F jS, Y h:i A', $post['Post']['created'], null, $user['User']['ti
// Affichera August 22nd, 2011 11:53 PM pour un utilisateur dans GMT+0
// August 22nd, 2011 03:53 PM pour un utilisateur dans GMT-8
// et August 23rd, 2011 09:53 AM GMT+10
```

La plupart des méthodes du Helper Time contiennent un paramètre $timezone. Le paramètre $timezone accepte une chaîne identifiante de timezone valide ou une instance de la classe *DateTimeZone*.

Formatage

`TimeHelper::convert`(*$serverTime*, *$timezone = NULL*)

Type retourné integer

Convertit étant donné le time (dans le time zone du serveur) vers le time de l'utilisateur, étant donné son/sa sortie de GMT. :

```
// appel via TimeHelper
echo $this->Time->convert(time(), 'Asia/Jakarta');
// 1321038036

// appel avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::convert(time(), new DateTimeZone('Asia/Jakarta'));
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.

`TimeHelper::convertSpecifiers`(*$format*, *$time = NULL*)

Type retourné string

Convertit une chaîne de caractères représentant le format pour la fonction strftime et retourne un format windows safe et i18n aware.

`TimeHelper::dayAsSql`(*$dateString*, *$field_name*, *$timezone = NULL*)

Type retourné string

Crée une chaîne de caractères dans le même format que dayAsSql mais nécessite seulement un unique objet date :

```
// Appelé avec TimeHelper
echo $this->Time->dayAsSql('Aug 22, 2011', 'modified');
// (modified >= '2011-08-22 00:00:00') AND (modified <= '2011-08-22 23:59:59')

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::dayAsSql('Aug 22, 2011', 'modified');
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`TimeHelper::daysAsSql`(*$begin*, *$end*, *$fieldName*, *$userOffset* = *NULL*)

Type retourné string

Retourne une chaîne de caractères dans le format "($field_name >= '2008-01-21 00 :00 :00') AND ($field_name <= '2008-01-25 23 :59 :59')". C'est pratique si vous avez besoin de chercher des enregistrements entre deux dates incluses :

```
// Appelé avec TimeHelper
echo $this->Time->daysAsSql('Aug 22, 2011', 'Aug 25, 2011', 'created');
// (created >= '2011-08-22 00:00:00') AND (created <= '2011-08-25 23:59:59')

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::daysAsSql('Aug 22, 2011', 'Aug 25, 2011', 'created');
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`TimeHelper::format`(*$date*, *$format* = *NULL*, *$default* = *false*, *$timezone* = *NULL*)

Type retourné string

Va retourner une chaîne formatée avec le format donné en utilisant les options de formatage de la fonction PHP date() (http ://www.php.net/manual/en/function.date.php) :

```
// appel via TimeHelper
echo $this->Time->format('F jS, Y h:i A', '2011-08-22 11:53:00');
// August 22nd, 2011 11:53 AM

echo $this->Time->format('r', '+2 days');
// 2 days from now formatted as Sun, 13 Nov 2011 03:36:10 +0800

// appel avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::format('2011-08-22 11:53:00', 'F jS, Y h:i A');
echo CakeTime::format('+2 days', 'r');
```

Vous pouvez aussi fournir la date/time en premier argument. En faisant cela vous devrez utiliser le format `strftime` compatible. Cette signature d'appel vous permet de tirer parti du format de date de la locale ce qui n'est pas possible en utilisant le format de `date()` compatible :

```
// appel avec TimeHelper
echo $this->Time->format('2012-01-13', '%d-%m-%Y', 'invalid');

// appel avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::format('2011-08-22', '%d-%m-%Y');
```

Modifié dans la version 2.2 : Les paramètres `$format` et `$date` sont en ordre opposé par rapport à ce qui se faisait dans 2.1 et suivants. Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants. Le paramètre `$default` remplace le paramètre

`$invalid` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$date` accepte aussi maintenant un objet DateTime.

`TimeHelper::fromString`(*$dateString*, *$timezone = NULL*)

Type retourné string

Prend une chaîne et utilise strtotime (http ://us.php.net/manual/en/function.date.php) pour la convertir en une date integer :

```
// Appelé avec TimeHelper
echo $this->Time->fromString('Aug 22, 2011');
// 1313971200

echo $this->Time->fromString('+1 days');
// 1321074066 (+1 day from current date)

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::fromString('Aug 22, 2011');
echo CakeTime::fromString('+1 days');
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`TimeHelper::gmt`(*$dateString = NULL*)

Type retourné integer

Va retourner la date en un nombre défini sur Greenwich Mean Time (GMT). :

```
// Appelé avec TimeHelper
echo $this->Time->gmt('Aug 22, 2011');
// 1313971200

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::gmt('Aug 22, 2011');
```

`TimeHelper::i18nFormat`(*$date*, *$format = NULL*, *$invalid = false*, *$timezone = NULL*)

Type retourné string

Retourne une chaîne de date formatée, étant donné soit un timestamp UNIX soit une chaîne de date valide strtotime(). Il prend en compte le format de la date par défaut pour le langage courant si un fichier LC_TIME est utilisé. Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.

`TimeHelper::nice`(*$dateString = NULL*, *$timezone = NULL*, *$format = null*)

Type retourné string

Prend une chaîne de date et la sort au format "Tue, Jan 1st 2008, 19 :25" ou avec le param optionnel `$format` :

```
// Appelé avec TimeHelper
echo $this->Time->nice('2011-08-22 11:53:00');
// Mon, Aug 22nd 2011, 11:53

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::nice('2011-08-22 11:53:00');
```

`TimeHelper::niceShort`(*$dateString = NULL*, *$timezone = NULL*)

Type retourné string

Prend une chaîne de date et la sort au format "Jan 1st 2008, 19 :25". Si l'objet date est today, le format sera "Today, 19 :25". Si l'objet date est yesterday, le format sera "Yesterday, 19 :25" :

```
// Appelé avec TimeHelper
echo $this->Time->niceShort('2011-08-22 11:53:00');
// Aug 22nd, 11:53

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::niceShort('2011-08-22 11:53:00');
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`TimeHelper::serverOffset`()

Type retourné integer

Retourne la valeur du serveur à partir du GMT dans les secondes.

`TimeHelper::timeAgoInWords`(*$dateString*, *$options = array()*)

Type retourné string

Prendra une chaîne datetime (tout ce qui est parsable par la fonction strtotime() de PHP ou le format de datetime de MySQL) et la convertit en un format de texte comme, "3 weeks, 3 days ago" :

```
// Appelé avec TimeHelper
echo $this->Time->timeAgoInWords('Aug 22, 2011');
// on 22/8/11

echo $this->Time->timeAgoInWords('Aug 22, 2011', array('format' => 'F jS, Y'));
// on August 22nd, 2011

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::timeAgoInWords('Aug 22, 2011');
echo CakeTime::timeAgoInWords('Aug 22, 2011', array('format' => 'F jS, Y'));
```

Utilisez l'option 'end' pour déterminer le point de cutoff pour ne plus utiliser de mots ; par défaut à '+1 month' :

```
// Appelé avec TimeHelper
echo $this->Time->timeAgoInWords('Aug 22, 2011', array('format' => 'F jS, Y', 'end' =>
// On Nov 10th, 2011 it would display: 2 months, 2 weeks, 6 days ago

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::timeAgoInWords('Aug 22, 2011', array('format' => 'F jS, Y', 'end' => '+
```

Utilisez l'option 'accuracy' pour déterminer la précision de la sortie. Vous pouvez utiliser ceci pour limiter la sortie :

```
// Si $timestamp est il y a 1 month, 1 week, 5 days et 6 hours
echo CakeTime::timeAgoInWords($timestamp, array(
    'accuracy' => array('month' => 'month'),
    'end' => '1 year'
));
// Sort '1 month ago'
```

Modifié dans la version 2.2 : L'option `accuracy` a été ajoutée.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`TimeHelper::toAtom`(*$dateString*, *$timezone = NULL*)

Type retourné string

Va retourner une chaîne de date au format Atom "2008-01-12T00 :00 :00Z" Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`TimeHelper::toQuarter`(*$dateString*, *$range = false*)

Type retourné mixed

Va retourner 1, 2, 3 ou 4 dépendant du quart de l'année sur lequel la date tombe. Si range est défini à true, un tableau à deux éléments va être retourné avec les dates de début et de fin au format "2008-03-31" :

```
// Appelé avec TimeHelper
echo $this->Time->toQuarter('Aug 22, 2011');
// Afficherait 3

$arr = $this->Time->toQuarter('Aug 22, 2011', true);
/*
Array
(
    [0] => 2011-07-01
    [1] => 2011-09-30
)
*/

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::toQuarter('Aug 22, 2011');
$arr = CakeTime::toQuarter('Aug 22, 2011', true);
```

Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.Introduit dans la version 2.4 : Les nouveaux paramètres d'option `relativeString` (par défaut à `%s ago`) et `absoluteString` (par défaut à `on %s`) pour permettre la personnalisation de la chaîne de sortie résultante sont maintenant disponibles.

`TimeHelper::toRSS`(*$dateString*, *$timezone = NULL*)

Type retourné string

Va retourner une chaîne de date au format RSS "Sat, 12 Jan 2008 00 :00 :00 -0500" Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`TimeHelper::toUnix`(*$dateString*, *$timezone = NULL*)

Type retourné integer

Un enrouleur pour fromString. Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`TimeHelper::toServer`(*$dateString*, *$timezone = NULL*, *$format = 'Y-m-d H :i :s'*)

Type retourné mixed

Introduit dans la version 2.2 : Retourne une date formatée dans le timezone du serveur.

`TimeHelper::timezone`(*$timezone = NULL*)

Type retourné DateTimeZone

Introduit dans la version 2.2 : Retourne un objet timezone à partir d'une chaîne de caractères ou de l'objet timezone de l'utilisateur. Si la fonction est appelée sans paramètres, elle essaie d'obtenir le timezone de la variable de configuration 'Config.timezone'.

`TimeHelper::listTimezones`(*$filter = null*, *$country = null*, *$group = true*)

Type retourné array

Introduit dans la version 2.2 : Retourne une liste des identificateurs de timezone.

Tester Time

`TimeHelper::isToday`(*$dateString*, *$timezone = NULL*)

`TimeHelper::isThisWeek`(*$dateString*, *$timezone = NULL*)

`TimeHelper::isThisMonth`(*$dateString*, *$timezone = NULL*)

`TimeHelper::isThisYear`(*$dateString*, *$timezone = NULL*)

`TimeHelper::wasYesterday`(*$dateString*, *$timezone = NULL*)

`TimeHelper::isTomorrow`(*$dateString*, *$timezone = NULL*)

`TimeHelper::isFuture`(*$dateString*, *$timezone = NULL*)

Introduit dans la version 2.4.

`TimeHelper::isPast`(*$dateString*, *$timezone = NULL*)

Introduit dans la version 2.4.

`TimeHelper::wasWithinLast` (*$timeInterval*, *$dateString*, *$timezone = NULL*)

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime. Toutes les fonctions ci-dessus retourneront true ou false quand une chaîne de date est passé. `wasWithinLast` prend une option supplémentaire `$time_interval` :

```
// Appelé avec TimeHelper
$this->Time->wasWithinLast($time_interval, $dateString);

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
CakeTime::wasWithinLast($time_interval, $dateString);
```

`wasWithinLast` prend un intervalle de time qui est une chaîne au format "3 months" et accepte un intervalle de time en secondes, minutes, heures, jours, semaines, mois et années (pluriels ou non). Si un intervalle de time n'est pas reconnu (par exemple, si il y a une faute de frappe) ensuite ce sera par défaut à days.

Utilitaires

En plus des components du coeur MVC, CakePHP inclut une bonne sélection de classes utilitaires qui vous aident à faire tout à partir de requêtes de services web, de mettre en cache, de se logger, d'internationaliser et plus encore.

Utilitaires

La mise en cache

La mise en cache est fréquemment utilisée pour réduire le temps pris pour créer ou lire depuis une autre ressource. La mise en cache est souvent utilisée pour rendre la lecture de ressources consommatrices en temps en ressources moins consommatrices. Vous pouvez aisément stocker le résultats de requêtes consommatrices en ressources, ou des accès à distance a des services web qui ne changent pas fréquemment dans un cache. Une fois dans un cache, re-lire les ressources stockées depuis le cache est moins consommateur en ressource qu'un accès a une ressource distante.

Mettre en cache dans CakePHP est principalement facilité par la classe `Cache`. Cette classe fournit un ensemble de méthodes statiques qui fournissent une API uniforme pour le traitement des différentes implémentations de mise en cache. CakePHP arrive avec plusieurs moteurs de cache intégrés, et fournit un système facile pour implémenter votre propre système de mise en cache. Les moteurs de cache intégrés sont :

- `FileCache` File cache est un cache simple qui utilise des fichiers locaux. C'est le moteur de cache le plus lent, et ne fournit que peut de fonctionnalité pour les opérations atomiques. Cependant, le stockage sur disque est souvent peu consommateur en ressource, au stockage de grands objets, ou des éléments qui sont rarement écrits fonctionnent bien dans les fichiers. C'est le moteur de Cache par défaut pour 2.3+.
- `ApcCache` Le cache APC utilise l'extension PHP APC (http ://php.net/apc). Cette extension utilise la mémoire partagée du serveur Web pour stocker les objets. Cela le rend très rapide, et capable de fournir les fonctionnalités atomiques en lecture/écriture. Par défaut CakePHP dans 2.0-2.2 utilisera ce moteur de cache si il est disponible.

- `Wincache` utilise l'extension Wincache (http ://php.net/wincache). Wincache est similaire aux fonctionnalités APC au niveau des fonctionnalités et des performances, mais optimisé pour Windows et IIS.
- `XcacheEngine` Similaire à APC, Xcache (http ://xcache.lighttpd.net/). est une extension PHP qui fournit des fonctionnalités similaires à APC.
- `MemcacheEngine` Utilise l'extension Memcache (http ://php.net/memcache). Memcache fournit un cache très rapide qui peut être distribué au travers de nombreux serveurs, et fournit les opérations atomiques.
- `RedisEngine` Utilise l'extension phpredis (https ://github.com/nicolasff/phpredis). Redis fournit un système de cache cohérent et rapide similaire à memcached, et fournit aussi des opérations atomic.

. versionchanged : : 2.3 FileEngine est toujours le moteur de cache par défaut. Dans le passé, un certain nombre de personnes avait des difficultés à configurer et déployer APC correctement dans les deux cli + web. Utiliser les fichiers devrait faciliter la configuration de CakePHP pour les nouveaux développeurs.

Quelque soit le moteur de cache que vous choisirez d'utiliser, votre application interagit avec `Cache` de manière cohérente. Cela signifie que vous pouvez aisément permuter les moteurs de cache en fonction de l'évolution de votre application. En plus de la classe `Cache`, le Helper *CacheHelper* vous permet le cache en pleine page, qui peut ainsi grandement améliorer les performances.

Configuration de la classe Cache

La configuration de la classe Cache peut être effectuée n'importe où, mais généralement vous voudrez configurer le cache dans `app/Config/bootstrap.php`. Vous pouvez configurer autant de configurations de cache dont vous avez besoin, et utiliser tous les mélanges de moteurs de cache. CakePHP utilise deux configurations de cache en interne, qui sont configurés dans `app/Config/core.php`. Si vous utilisez APC ou Memcache vous devrez vous assurer de définir des clés uniques pour les caches du noyau. Ceci vous évitera que de multiples applications viennent réécrire les données cache de l'autre.

L'utilisation de multiples configurations de cache peut aider à réduire le nombre de fois où vous aurez à utiliser `Cache::set()`. Aussi bien que centraliser tout vos paramètres de cache. L'utilisation de configurations multiples vous permet également de changer le stockage comme vous l'entendez.

Note : Vous devez spécifier quel moteur utiliser. Il ne met **pas** par défaut à *File*.

Exemple :

```
Cache::config('short', array(
    'engine' => 'File',
    'duration' => '+1 hours',
    'path' => CACHE,
    'prefix' => 'cake_short_'
));

// long
Cache::config('long', array(
    'engine' => 'File',
    'duration' => '+1 week',
```

```
    'probability' => 100,
    'path' => CACHE . 'long' . DS,
));
```

En insérant le code ci-dessus dans votre `app/Config/bootstrap.php` vous aurez deux configurations de cache supplémentaires. Le nom de ces configurations 'short' ou 'long' est utilisé comme le paramètre `$config` pour `Cache::write()` et `Cache::read()`.

Note : Quand vous utilisez le moteur FileEngine vous pourriez avoir besoin de l'option `mask` pour vous assurer que les fichiers cachés sont créés avec les bonnes permissions.

Création d'un moteur de stockage pour le Cache

Vous pouvez fournir vos propre adaptateurs `Cache` dans `app/Lib` aussi bien que dans un plugin en utilisant `$plugin/Lib`. Les moteurs de cache App/plugin peuvent aussi remplacer les moteurs du noyau. Les adaptateurs de cache doivent être dans un répertoire cache. Si vous avez un moteur de cache nommé `MonMoteurDeCachePerso` il devra être placé soit dans `app/Lib/Cache/Engine/MonMoteurDeCachePerso.php` comme une app/libs. Ou dans `$plugin/Lib/Cache/Engine/MonMoteurDeCachePerso.php` comme parti d'un plugin. Les configurations de cache provenant de plugin doivent utiliser la notation par points de plugin. :

```
Cache::config('custom', array(
    'engine' => 'CachePack.MyCustomCache',
    // ...
));
```

Note : Le cache App et plugin doit être configuré dans `app/Config/bootstrap.php`. Si vous essayez de les configurer dans core.php ils ne fonctionneront pas correctement.

Les moteurs de cache personnalisés doivent entendre `CacheEngine` qui définit un nombre de méthodes d'abstraction ainsi que quelques méthodes d'initialisation.

L'API requise pour CacheEngine est

class `CacheEngine`

La classe de base pour tous les moteurs de cache utilisé avec le Cache.

`CacheEngine::write`(*$key*, *$value*, *$config = 'default'*)

Retourne un booléen en cas de succès.

Écrit la valeur d'une clé dans le cache, la chaîne optionnelle $config spécifie le nom de la configuration à écrire.

`CacheEngine::read`(*$key*)

Retourne La valeur cachée ou false en cas d'échec.

Lit une clé depuis le cache. Retourne false pour indiquer que l'entrée a expiré ou n'existe pas.

`CacheEngine::delete`(*$key*)

Retourne Un booléen true en cas de succès.

Efface une clé depuis le cache. Retourne false pour indiquer que l'entrée n'existe pas ou ne peut être effacée.

`CacheEngine::clear`(*$check*)

Retourne Un Booléen true en cas de succès.

Efface toutes les clés depuis le cache. Si $check est true, vous devez valider que chacune des valeurs est actuellement expirée.

`CacheEngine::clearGroup`(*$group*)

Retourne Boolean true on success.

Supprime toutes les clés à partir du cache appartenant au même groupe.

`CacheEngine::decrement`(*$key*, *$offset = 1*)

Retourne Un boléen true en cas de succès.

Décrémente un nombre dans la clé et retourne la valeur décrémentée

`CacheEngine::increment`(*$key*, *$offset = 1*)

Retourne Un boléen true en cas de succès.

Incrémente un nombre dans la clé et retourne la valeur incrémentée

`CacheEngine::gc()`

Non requis, mais utilisé pour faire du nettoyage quand les ressources expires. Le moteur FileEngine utilise cela pour effacer les fichiers qui contiennent des contenus expirés.

Utilisation du Cache pour stocker le résultat des requêtes les plus courantes

Vous pouvez considérablement améliorer les performances de vos applications en plaçant les résultats qui ne changent que peu fréquemment, ou qui peuvent être sujets à de nombreuses lectures dans le cache. Un exemple parfait de ceci sont les résultats d'un find `Model::find()`. Une méthode qui utilise Le Cache pour stocker les résultats pourrait ressembler à cela

```
class Post extends AppModel {

    public function newest() {
        $result = Cache::read('newest_posts', 'longterm');
        if (!$result) {
            $result = $this->find('all', array('order' => 'Post.updated DESC', 'limit' => 1
            Cache::write('newest_posts', $result, 'longterm');
        }
        return $result;
    }
}
```

Vous pouvez améliorer le code ci-dessus en déplaçant la lecture du cache dans un comportement, qui lit depuis le cache, ou qui exécute les méthodes de model. C'est un exercice que vous pouvez faire.

Utilisation du Cache pour stocker les compteurs

L'utilisation de compteurs dans le cache peut être une chose intéressante. Par exemple un simple compte à rebours pour retenir les 'slots' restants d'un concours pourraient être stockés en Cache. La classe Cache propose des moyens atomiques pour incrémenter/décrémenter des valeurs de compteur facilement. Les opérations atomiques sont importantes pour ces valeurs parce que ça réduit le risque de contention, et la capacité de deux utilisateurs à simultanément en abaisser la valeur et de résulter à une valeur incorrecte.

Après avoir défini une valeur entière vous pouvez la manipuler en utilisant `Cache::increment()` et `Cache::decrement()` :

```
Cache::write('compteur_initial', 10);

// Plus tard sur
Cache::decrement('compteur_initial');

//ou
Cache::increment('compteur_initial');
```

Note : L'incrémentation et la décrémentation ne fonctionne pas avec le moteur FileEngine. Vous devez utiliser APC ou Memcache en remplacement.

Utilisation des groupes

Introduit dans la version 2.2. Parfois vous voudrez marquer plusieurs entrées de cache comme appartenant à un même groupe ou un namespace. C'est une exigence courante pour invalider des grosses quantités de clés alors que quelques changements d'informations sont partagés pour toutes les entrées dans un même groupe. Cela est possible en déclarant les groupes dans la configuration deu cache :

```
Cache::config('site_home', array(
    'engine' => 'Redis',
    'duration' => '+999 days',
    'groups' => array('comment', 'post')
));
```

Disons que vous voulez stocker le HTML généré pour votre page d'accueil dans le cache, mais voulez aussi invalider automatiquement ce cache à chaque fois qu'un commentaire ou un post est ajouté à votre base de données. En ajoutant les groupes `comment` et `post`, nous avons effectivement taggés les clés stockées dans la configuration du cache avec les noms des deux groupes.

Par exemple, dès qu'un post est ajouté, nous pouvons dire au moteur de Cache de retirer toutes les entrées associées au groupe `post` :

```
// Model/Post.php

public function afterSave($created) {
    if ($created) {
        Cache::clearGroup('post', 'site_home');
```

```
    }
}
```

Introduit dans la version 2.4. `Cache::groupConfigs()` peut être utilisée pour récupérer les correspondances entre le groupe et les configurations, par ex : en ayant le même groupe :

```
// Model/Post.php

/**
 * Une variation de l\'exemple précédent qui nettoie toutes les
 * configurations de Cache ayant le même groupe
 */
public function afterSave($created) {
    if ($created) {
        $configs = Cache::groupConfigs('post');
        foreach ($configs['post'] as $config) {
            Cache::clearGroup('post', $config);
        }
    }
}
```

Les groupes son partagés à travers toutes les configs de cache en utilisant le même moteur et le même préfixe. Si vous utilisez les groupes et voulez tirer profit de la suppression de groupe, choisissez un préfixe commun pour toutes vos configs.

l'API de Cache

class `Cache`

La classe Cache dans CakePHP fournit un frontend générique pour plusieurs systèmes de cache backend. Différentes configurations de Cache et de moteurs peuvent être configurés dans votre app/Config/core.php

static `Cache::config`(*$name = null*, *$settings = array()*)

`Cache::config()` est utilisée pour créer des configurations de cache supplémentaire. Ces configurations supplémentaires peuvent avoir différentes durées, moteurs, chemins, ou préfixes que la configuration par défaut du cache.

static `Cache::read`(*$key*, *$config = 'default'*)

Cache : :read() est utilisée pour lire la valeur en cache stockée dans `$key` depuis le `$config`. Si $config est null la configuration par défaut sera utilisée. `Cache::read()` retournera la valeur en cache si c'est un cache valide ou `false` si le cache a expiré ou n'existe pas. Le contenu du cache pourrait évaluer false, donc soyez sûr que vous utilisez l'opérateur de comparaison stricte === ou !==.

Par exemple :

```
$cloud = Cache::read('cloud');

if ($cloud !== false) {
    return $cloud;
}
```

```
// génération des données cloud
// ...

// stockage des donnée en cache
Cache::write('cloud', $cloud);
return $cloud;
```

static `Cache::write` (*$key*, *$value*, *$config = 'default'*)

Cache : :write() Ecrira $value dans le cache. Vous pouvez lire ou effacer cette valeur plus tard en vous y référant avec `$key`.. Vous pouvez spécifier une configuration optionnelle pour stocker le cache. Si il n'y a pas de `$config` de spécifié c'est la configuration par défaut qui sera appliquée. Cache : :write() peut stocker n'importe quel type d'objet est est idéal pour stocker les résultats des finds de vos modèles. :

```
if (($posts = Cache::read('posts')) === false) {
    $posts = $this->Post->find('all');
    Cache::write('posts', $posts);
}
```

Utiliser `Cache::write()` et `Cache::read()` pour aisément réduire le nombre de déplacement fait dans la base de données pour rechercher les posts.

static `Cache::delete` (*$key*, *$config = 'default'*)

`Cache::delete()` vous permet d'enlever complètement un objet en cache du lieu de stockage.

static `Cache::set` (*$settings = array()*, *$value = null*, *$config = 'default'*)

`Cache::set()` vous permet de réécrire temporairement les paramètres de configs pour une opération (habituellement une lecture ou écriture). Si vous utilisez `Cache::set()` pour changer les paramètres pour une écriture, vous devez aussi utiliser `Cache::set()` avant de lire les données en retour. Si vous ne faites pas cela, les paramètres par défaut seront utilisés quand la clé de cache est lu. :

```
Cache::set(array('duration' => '+30 days'));
Cache::write('results', $data);

// plus tard

Cache::set(array('duration' => '+30 days'));
$results = Cache::read('results');
```

Si vous trouvez que vous répétez l'appel à `Cache::set()` peut être devriez-vous créer une nouvelle `Cache::config()`. Qui enlèvera les besoins d'appeler `Cache::set()`.

static `Cache::increment` (*$key*, *$offset = 1*, *$config = 'default'*)

Incrémente de manière atomique une valeur stockée dans le moteur de cache. Idéal pour modifier un compteur ou des valeurs de sémaphore.

static `Cache::decrement` (*$key*, *$offset = 1*, *$config = 'default'*)

Décrémente de manière atomique une valeur stockée dans le moteur de cache. Idéal pour modifier un compteur ou des valeurs de sémaphore.

static `Cache::clear` (*$check*, *$config = 'default'*)

Détruit toutes les valeurs en cache pour une configuration de cache. Dans les moteurs comme Apc,

Memcache et Wincache le préfixe de configuration de cache est utilisé pour enlever les entrées de cache. Soyez sûre que différentes configuration de cache ont différent préfixe.

`Cache::clearGroup`(*$group*, *$config = 'default'*)

Retourne Boléen true en cas de succès.

Supprime toutes les clés du cache appartenant au même groupe.

static `Cache::gc`(*$config*)

collectes d'entrée de Garbage dans la configuration du cache. Utilisé principalement par FileEngine. Il devrait être mis en œuvre par n'importe quel moteur de cache qui requiert des évictions manuelles de données en cache.

static `Cache::groupConfigs`(*$group = null*)

Retourne Tableau de groups et leurs noms de configuration liés.

Récupère les noms de group pour configurer la coorespondance.

CakeEmail

class `CakeEmail`(*mixed $config = null*)

`CakeEmail` est une nouvelle classe pour envoyer des emails. Avec cette classe, vous pouvez envoyer des emails de n'importe quel endroit de votre application. En plus d'utiliser le EmailComponent à partir de vos controllers, vous pouvez aussi envoyer des mails à partir des Shells et des Models.

Cette classe remplace `EmailComponent` et donne plus de flexibilité dans l'envoi d'emails. Par exemple, vous pouvez créer vos propres transports pour envoyer l'email au lieu d'utiliser les smtp et mail fournis.

Utilisation basique

Premièrement, vous devez vous assurer que la classe est chargée en utilisant `App::uses()` :

```
App::uses('CakeEmail', 'Network/Email');
```

L'utilisation de CakeEmail est similaire à l'utilisation de `EmailComponent`. Mais au lieu d'utiliser les attributs, vous devez utiliser les méthodes. Exemple :

```
$Email = new CakeEmail();
$Email->from(array('me@example.com' => 'My Site'));
$Email->to('you@example.com');
$Email->subject('About');
$Email->send('My message');
```

Pour simplifier les choses, toutes les méthodes de setter retournent l'instance de classe. Vous pouvez réécrire le code ci-dessous :

```
$Email = new CakeEmail();
$Email->from(array('me@example.com' => 'My Site'))
    ->to('you@example.com')
    ->subject('About')
    ->send('Mon message');
```

Choisir l'emetteur Quand on envoie des emails de la part d'autre personne, c'est souvent une bonne idée de définir l'emetteur original en utilisant le header Sender. Vous pouvez faire ceci en utilisant `sender()`

```
$Email = new CakeEmail();
$Email->sender('app@example.com', 'MyApp emailer');
```

Note : C'est aussi une bonne idée de définir l'envelope de l'emetteur quand on envoie un mail de la part d'une autre personne. Cela les empêche d'obtenir tout message sur la délivrance.

Configuration

La configuration est la même que pour la base de données, les emails peuvent avoir une classe pour centraliser toute la configuration.

Créer le fichier `app/Config/email.php` avec la classe `EmailConfig`. Le fichier `app/Config/email.php.default` donne un exemple de ce fichier.

`CakeEmail` va créer une instance de la classe `EmailConfig` pour accéder à config. Si vous avez des données dynamiques à mettre dans les configs, vous pouvez utiliser le constructeur pour le faire :

```
class EmailConfig {
    public function __construct() {
        // Faire des assignments conditionnel ici.
    }
}
```

Il n'est pas nécéssaire de créer `app/Config/email.php`, `CakeEmail` peut être utilisé sans lui et utiliser les méthodes respectives pour définir toutes les configurations séparément ou charger un tableau de configs.

Pour charger un config à partir de `EmailConfig`, vous pouvez utiliser la méthode `config()` ou la passer au constructeur de `CakeEmail` :

```
$Email = new CakeEmail();
$Email->config('default');

//ou dans un constructeur::
$Email = new CakeEmail('default');
```

Plutôt que de passer une chaîne qui correspond au nom de la configuration dans `EmailConfig`, vous pouvez aussi juste charger un tableau de configs :

```
$Email = new CakeEmail();
$Email->config(array('from' => 'me@example.org', 'transport' => 'MyCustom'));

//ou dans un constructeur::
$Email = new CakeEmail(array('from' => 'me@example.org', 'transport' => 'MyCustom'));
```

Vous pouvez configurer les serveurs SSL SMTP, comme Gmail. Pour faire ceci, mettez `'ssl://'` en préfixe dans le host et configurez la valeur du port selon. Exemple :

```
class EmailConfig {
    public $gmail = array(
        'host' => 'ssl://smtp.gmail.com',
        'port' => 465,
        'username' => 'my@gmail.com',
        'password' => 'secret',
        'transport' => 'Smtp'
    );
}
```

Note : Pour utiliser cette fonctionnalité, vous aurez besoin d'avoir SSL configuré dans votre installation PHP.

Depuis 2.3.0, vous pouvez aussi activer TLS SMTP en utilisant l'option `tls` :

```
class EmailConfig {
    public $gmail = array(
        'host' => 'smtp.gmail.com',
        'port' => 465,
        'username' => 'my@gmail.com',
        'password' => 'secret',
        'transport' => 'Smtp',
        'tls' => true
    );
}
```

La configuration ci-dessus va activer la communication TLS pour les messages emails.

Configurations La clés de configuration suivantes sont utilisées :

- `'from'` : Email ou un tableau d'emmeteur. Regardez `CakeEmail::from()`.
- `'sender'` : Email ou un tableau d'emetteur réel. Regardez `CakeEmail::sender()`.
- `'to'` : Email ou un tableau de destination. Regardez `CakeEmail::to()`.
- `'cc'` : Email ou un tableau de copy carbon. Regardez `CakeEmail::cc()`.
- `'bcc'` : Email ou un tableau de copy carbon blind. Regardez `CakeEmail::bcc()`.
- `'replyTo'` : Email ou un tableau de repondre à cet e-mail. Regardez `CakeEmail::replyTo()`.
- `'readReceipt'` : Adresse Email ou un tableau d'adresses pour recevoir un récepissé de lecture. Regardez `CakeEmail::readReceipt()`.
- `'returnPath'` : Adresse Email ou un tableau des adresses à retourner si vous avez une erreur. Regardez `CakeEmail::returnPath()`.
- `'messageId'` : ID du Message de l'e-mail. Regardez `CakeEmail::messageId()`.
- `'subject'` : Sujet du message. Regardez `CakeEmail::subject()`.
- `'message'` : Contenu du message. Ne définissez pas ce champ si vous utilisez un contenu rendu.
- `'headers'` : Headers à inclure. Regardez `CakeEmail::setHeaders()`.
- `'viewRender'` : Si vous utilisez un contenu rendu, définissez le nom de classe de la vue. Regardez `CakeEmail::viewRender()`.
- `'template'` : Si vous utilisez un contenu rendu, définissez le nom du template. Regardez `CakeEmail::template()`.
- `'theme'` : Theme utilisé pour le rendu du template. Voir `CakeEmail::theme()`.

- `'layout'` : Si vous utilisez un contenu rendu, définissez le layout à rendre. Si vous voulez rendre un template sans layout, définissez ce champ à null. Regardez `CakeEmail::template()`.
- `'viewVars'` : Si vous utilisez un contenu rendu, définissez le tableau avec les variables devant être rendus dans la vue. Regardez `CakeEmail::viewVars()`.
- `'attachments'` : Liste des fichiers à attacher. Regardez `CakeEmail::attachments()`.
- `'emailFormat'` : Format de l'email (html, text ou both). Regardez `CakeEmail::emailFormat()`.
- `'transport'` : Nom du Transport. Regardez `CakeEmail::transport()`.
- `'log'` : Niveau de Log pour connecter les headers del'email headers et le message. `true` va utiliser LOG_DEBUG. Regardez aussi `CakeLog::write()`

Toutes ces configurations sont optionnelles, excepté `'from'`. Si vous mettez plus de configuration dans ce tableau, les configurations seront utilisées dans la méthode `CakeEmail::config()` et passées à la classe de transport `config()`. Par exemple, si vous utilisez le transport smtp, vous devez passer le host, port et autres configurations.

Note : Les valeurs des clés ci-dessus utilisant Email ou un tableau, comme from, to, cc etc. seront passées en premier paramètre des méthodes correspondantes. L'equivalent pour `CakeEmail::from('my@example.com', 'My Site')` sera défini comme `'from' => array('my@example.com' => 'My Site')` dans votre config.

Définir les headers Dans `CakeEmail`, vous êtes libre de définir les headers que vous souhaitez. Si vous migrez pour utiliser CakeEmail, n'oubliez pas de mettre le préfixe `X-` dans vos headers.

Regardez `CakeEmail::setHeaders()` et `CakeEmail::addHeaders()`

Envoyer les emails templatés Les Emails sont souvent bien plus que de simples message textes. Afin de faciliter cela, CakePHP fournit une façon d'envoyer les emails en utilisant la *view layer* de CakePHP.

Les templates pour les emails se placent dans un dossier spécial dans le répertoire `View` de votre application. Les vues des emails peuvent aussi utiliser les layouts et éléments tout comme les vues normales :

```
$Email = new CakeEmail();
$Email->template('welcome', 'fancy')
    ->emailFormat('html')
    ->to('bob@example.com')
    ->from('app@domain.com')
    ->send();
```

Ce qui est au-dessus utilise `app/View/Emails/html/welcome.ctp` pour la vue, et `app/View/Layouts/Emails/html/fancy.ctp` pour le layout. Vous pouvez aussi envoyer des messages email templaté multipart :

```
$Email = new CakeEmail();
$Email->template('welcome', 'fancy')
    ->emailFormat('both')
    ->to('bob@example.com')
    ->from('app@domain.com')
    ->send();
```

Ceci utiliserait les fichiers de vue suivants :
- `app/View/Emails/text/welcome.ctp`
- `app/View/Layouts/Emails/text/fancy.ctp`
- `app/View/Emails/html/welcome.ctp`
- `app/View/Layouts/Emails/html/fancy.ctp`

Quand on envoie les emails templatés, vous avez la possibilité d'envoyer soit `text`, `html` soit `both`.

Vous pouvez définir des variables de vue avec `CakeEmail::viewVars()` :

```
$Email = new CakeEmail('templated');
$Email->viewVars(array('value' => 12345));
```

Dans votre email template, vous pouvez utiliser ceux-ci avec :

```
<p>Ici est votre valeur: <b><?php echo $value; ?></b></p>
```

Vous pouvez aussi utiliser les helpers dans les emails, un peu comme vous pouvez dans des fichiers normaux de vue. Par défaut, seul `HtmlHelper` est chargé. Vous pouvez chargez des helpers supplémentaires en utilisant la méthode `helpers()` :

```
$Email->helpers(array('Html', 'Custom', 'Text'));
```

Quand vous définissez les helpers, assurez vous d'inclure 'Html' ou il sera retiré des helpers chargés dans votre template d'email.

Si vous voulez envoyer un email en utilisant templates dans un plugin, vous pouvez utiliser la *syntaxe de plugin* familière pour le faire :

```
$Email = new CakeEmail();
$Email->template('Blog.new_comment', 'Blog.auto_message')
```

Ce qui est au-dessus utiliserait les templates à partir d'un plugin de Blog par exemple.

Envoyer les pièces jointes Vous pouvez aussi attacher des fichiers aux messages d'email. Il y a quelques formats différents qui dépendent de quel type de fichier vous avez, et comment vous voulez que les noms de fichier apparaissent dans le mail de réception du client :

1. Chaîne de caractères : `$Email->attachments('/full/file/path/file.png')` va attacher ce fichier avec le nom file.png.
2. Tableau : `$Email->attachments(array('/full/file/path/file.png')` aura le même comportement qu'en utilisant une chaîne de caractères.
3. Tableau avec clé : `$Email->attachments(array('photo.png' => '/full/some_hash.png'))` va attacher some_hash.png avec le nom photo.png. Le récipiendaire va voir photo.png, pas some_hash.png.
4. Tableaux imbriqués :

```
    $Email->attachments(array(
        'photo.png' => array(
            'file' => '/full/some_hash.png',
            'mimetype' => 'image/png',
            'contentId' => 'my-unique-id'
```

```
        )
    ));
```

Ce qui est au-dessus va attacher le fichier avec différent mimetype et avec un content ID personnalisé (Quand vous définissez le content ID, la pièce jointe est transformée en inline). Le mimetype et contentId sont optionels dans ce formulaire.

4.1. Quand vous utilisez `contentId`, vous pouvez utiliser le fichier dans corps html comme `<img src="cid:my-content-id">`.

4.2. Vous pouvez utiliser l'option `contentDisposition` pour désactiver le header `Content-Disposition` pour une pièce jointe. C'est utile pour l'envoi d'invitations ical à des clients utilisant outlook.

4.3 Au lieu de l'option `file`, vous pouvez fournir les contenus de fichier en chaîne en utilisant l'option `data`. Cela vous permet d'attacher les fichiers sans avoir besoin de chemins de fichier vers eux.

Modifié dans la version 2.3 : L'option `contentDisposition` a été ajoutée.Modifié dans la version 2.4 : L'option `data` a été ajoutée.

Utiliser les transports Les Transports sont des classes destinés à envoyer l'email selon certain protocoles ou méthodes. CakePHP supporte les transports Mail (par défaut), Debug et Smtp.

Pour configurer votre méthode, vous devez utiliser la méthode `CakeEmail::transport()` ou avoir transport dans votre configuration.

Créer des Transports personnalisés Vous pouvez créer vos transports personnalisés pour intégrer avec d'autres systèmes email (comme SwiftMailer). Pour créer votre transport, créez tout d'abord le fichier `app/Lib/Network/Email/ExampleTransport.php` (où Exemple est le nom de votre transport). Pour commencer, votre fichier devrait ressembler à cela :

```
App::uses('AbstractTransport', 'Network/Email');

class ExempleTransport extends AbstractTransport {

    public function send(CakeEmail $Email) {
        // magique à l'intérieur!
    }

}
```

Vous devez intégrer la méthode `send(CakeEmail $Email)` avec votre logique personnalisée. En option, vous pouvez intégrer la méthode `config($config)`. `config()` est appelé avant send() et vous permet d'accepter les configurations de l'utilisateur. Par défaut, cette méthode met la configuration dans l'attribut protégé `$_config`.

Si vous avez besoin d'appeler des méthodes supplémentaires sur le transport avant l'envoi, vous pouvez utiliser `CakeEmail::transportClass()` pour obtenir une instance du transport. Exemple :

```
$yourInstance = $Email->transport('your')->transportClass();
$yourInstance->myCustomMethod();
$Email->send();
```

Faciliter les règles de validation des adresses

`CakeEmail::emailPattern($pattern = null)`

Si vous avez des problèmes de validation lors de l'envoi vers des adresses non conformes, vous pouvez faciliter le patron utilisé pour valider les adresses email. C'est parfois nécessaire quand il s'agit de certains ISP Japonais.

> $email = new CakeEmail('default') ;
>
> // Relax le patron d'email, ainsi vous pouvez envoyer // vers des adresses non conformes $email->emailPattern($newPattern) ;

Introduit dans la version 2.4.

Envoyer des messages rapidement

Parfois vous avez besoin d'une façon rapide d'envoyer un email, et vous n'avez pas particulièrement envie en même temps de définir un tas de configuration. `CakeEmail::deliver()` est présent pour ce cas.

Vous pouvez créer votre configuration dans `EmailConfig`, ou utiliser un tableau avec toutes les options dont vous aurez besoin et utiliser la méthode statique `CakeEmail::deliver()`. Exemple :

```
CakeEmail::deliver('you@example.com', 'Subject', 'Message', array('from' => 'me@example.com
```

Cette méthode va envoyer un email à you@example.com (you@example.com), à partir de me@example.com (me@example.com) avec le sujet Subject et le contenu Message.

Le retour de `deliver()` est une instance de `CakeEmail` avec l'ensemble des configurations. Si vous ne voulez pas envoyer l'email maintenant, et souhaitez configurer quelques trucs avant d'envoyer, vous pouvez passer le 5ème paramètre à false.

Le 3ème paramètre est le contenu du message ou un tableau avec les variables (quand on utilise le contenu rendu).

Le 4ème paramètre peut être un tableau avec les configurations ou une chaîne de caractères avec le nom de configuration dans `EmailConfig`.

Si vous voulez, vous pouvez passer les to, subject et message à null et faire toutes les configurations dans le 4ème paramètre (en tableau ou en utilisant `EmailConfig`). Vérifiez la liste des *configurations* pour voir toutes les configs acceptées.

Envoyer des emails depuis CLI

Modifié dans la version 2.2 : La méthode `domain()` a été ajoutée dans 2.2 Quand vous envoyez des emails à travers un script CLI (Shells, Tasks, ...), vous devez définir manuellement le nom de domaine que CakeEmail doit utiliser. Il sera utilisé comme nom d'hôte pour l'id du message (puisque il n'y a pas de nom d'hôte dans un environnement CLI) :

```
$Email->domain('www.example.org');
// Resulte en ids de message comme ``<UUID@www.example.org>`` (valid)
// au lieu de `<UUID@>`` (invalid)
```

Un id de message valide peut permettre à ce message de ne pas finir dans un dossier de spam.

CakeNumber

class CakeNumber

Si vous avez besoin des fonctionnalités de `NumberHelper` en-dehors d'une `View`, utilisez la classe `CakeNumber` :

```
class UsersController extends AppController {

    public $components = array('Auth');

    public function afterLogin() {
        App::uses('CakeNumber', 'Utility');
        $storageUsed = $this->Auth->user('storage_used');
        if ($storageUsed > 5000000) {
            // notify users of quota
            $this->Session->setFlash(__('You are using %s storage', CakeNumber::toReadableS
        }
    }
}
```

Introduit dans la version 2.1 : `CakeNumber` a été refondu à partir de `NumberHelper`. Toutes ces fonctions retournent le nombre formaté ; Elles n'affichent pas automatiqement la sortie dans la vue.

`CakeNumber::currency` (*mixed $number*, *string $currency = 'USD'*, *array $options = array()*)

Paramètres

- **$number** (*float*) – La valeur à convertir.
- **$currency** (*string*) – Le format de monnaie connu à utiliser.
- **$options** (*array*) – Options, voir ci-dessous.

Cette méthode est utilisée pour afficher un nombre dans des formats de monnaie courante (EUR,GBP,USD). L'utilisation dans une vue ressemble à ceci :

```
// Appelé avec NumberHelper
echo $this->Number->currency($number, $currency);

// Appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::currency($number, $currency);
```

Le premier paramètre $number, doit être un nombre à virgule qui représente le montant d'argent que vous désirez. Le second paramètre est utilisé pour choisir un schéma de formatage de monnaie courante :

$currency	1234.56, formaté par le type courant
EUR	€ 1.236,33
GBP	£ 1,236.33
USD	$ 1,236.33

Le troisième paramètre est un tableau d'options pour définir la sortie. Les options suivantes sont disponibles :

Option	Description
before	Le symbole de la monnaie à placer avant les nombres ex : '$'
after	Le symbole de la monnaie à placer après les nombres décimaux ex : 'c'. Définit le boléeen à false pour utiliser aucun symbole décimal ex : 0.35 => $0.35.
zero	Le texte à utiliser pour des valeurs à zéro, peut être une chaîne de caractères ou un nombre. ex : 0, 'Free !'
places	Nombre de décimales à utiliser. ex : 2
thousands	Séparateur des milliers ex : ','
decimals	Symbole de Séparateur des décimales. ex : '.'
negative	Symbole pour les nombres négatifs. Si égal à '()', le nombre sera entouré avec (et)
escape	La sortie doit-elle être échappée de htmlentity ? Par défaut défini à true
wholeSymbol	La chaîne de caractères à utiliser pour les tous nombres. ex : ' dollars'
wholePosition	Soit 'before' soit 'after' pour placer le symbole entier
fractionSymbol	Chaîne de caractères à utiliser pour les nombres en fraction. ex : ' cents'
fractionPosition	Soit 'before' soit 'after' pour placer le symbole de fraction

Si une valeur $currency non reconnue est fournie, elle est préfixée par un nombre formaté en USD. Par exemple :

```
// Appelé avec NumberHelper
echo $this->Number->currency('1234.56', 'FOO');

// Sortie
FOO 1,234.56

// Appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::currency('1234.56', 'FOO');
```

`CakeNumber::defaultCurrency`(*string $currency*)

Paramètres

- **$currency** (*string*) – Défini une monnaie connu pour `CakeNumber::currency()`.

Setter/getter pour la monnaie par défaut. Ceci retire la necessité de toujours passer la monnaie à `CakeNumber::currency()` et change toutes les sorties de monnaie en définissant les autres par défaut. Introduit dans la version 2.3 : Cette méthode a été ajoutée dans 2.3.

`CakeNumber::addFormat`(*string $formatName*, *array $options*)

Paramètres

- **$formatName** (*string*) – Le nom du format à utiliser dans le futur.
- **$options** (*array*) – Le tableau d'options pour ce format. Utilise les mêmes clés `$options` comme `CakeNumber::currency()`.
  - *before* Symbole de monnaie avant le nombre. False pour aucun.

- *after* Symbole de monnaie après le nombre. False pour aucun.
- *zero* Le texte à utiliser pour les valeurs à zéro, peut être une chaîne de caractères ou un nombre. ex : 0, 'Free !'
- *places* Nombre de décimal à utiliser. ex. 2.
- *thousands* Séparateur des milliers. ex : ','.
- *decimals* Symbole de Séparateur des Decimales. ex : '.'.
- *negative* Symbole pour les nombres négatifs. Si égal à '()', le nombre sera entouré avec (et).
- *escape* La sortie doit-elle être échappée de htmlentity ? Par défaut à true.
- *wholeSymbol* Chaîne de caractères à utiliser pour tous les nombres. ex : ' dollars'.
- *wholePosition* Soit 'before' soit 'after' pour placer le symbole complet.
- *fractionSymbol* Chaîne de caractères à utiliser pour les nombres à fraction. ex : ' cents'.
- *fractionPosition* Soit 'before' soit 'after' pour placer le symbole de fraction.

Ajoute le format de monnaie au helper Number. Facilite la réutilisation des formats de monnaie. :

```
// appelé par NumberHelper
$this->Number->addFormat('BRR', array('before' => 'R$ '));

// appelé par CakeNumber
App::uses('CakeNumber', 'Utility');
CakeNumber::addFormat('BRR', array('before' => 'R$ '));
```

Vous pouvez maintenant utiliser *BRR* de manière courte quand vous formattez les montants de monnaie :

```
// appelé par NumberHelper
echo $this->Number->currency($value, 'BRR');

// appelé par CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::currency($value, 'BRR');
```

Les formats ajoutés sont fusionnés avec les formats par défaut suivants :

```
array(
    'wholeSymbol'      => '',
    'wholePosition'    => 'before',
    'fractionSymbol'   => '',
    'fractionPosition' => 'after',
    'zero'             => 0,
    'places'           => 2,
    'thousands'        => ',',
    'decimals'         => '.',
    'negative'         => '()',
    'escape'           => true
)
```

`CakeNumber::precision`(*mixed $number*, *int $precision = 3*)

Paramètres

- **$number** (*float*) – La valeur à convertir
- **$precision** (*integer*) – Le nombre de décimal à afficher

Cette méthode affiche un nombre avec le montant de précision spécifié (place de la décimal). Elle arrondira afin de maintenir le niveau de précision défini. :

```
// appelé avec NumberHelper
echo $this->Number->precision(456.91873645, 2 );

// Sortie
456.92

// appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::precision(456.91873645, 2 );
```

`CakeNumber::toPercentage`(*mixed $number*, *int $precision = 2*, *array $options = array()*)

Paramètres

- **$number** (*float*) – La valeur à convertir.
- **$precision** (*integer*) – Le nomnbre de décimal à afficher.
- **$options** (*array*) – Options, voir ci-dessous.

Option	Description
multiply	Booléen pour indiquer si la valeur doit être multipliée par 100. Utile pour les pourcentages avec décimal.

Comme precision(), cette méthode formate un nombre selon la précision fournie (où les nombres sont arrondis pour parvenir à ce degré de précision). Cette méthode exprime aussi le nombre en tant que pourcentage et préfixe la sortie avec un signe de pourcent. :

```
// appelé avec NumberHelper. Sortie: 45.69%
echo $this->Number->toPercentage(45.691873645);

// appelé avec CakeNumber. Sortie: 45.69%
App::uses('CakeNumber', 'Utility');
echo CakeNumber::toPercentage(45.691873645);

// Appelé avec multiply. Sortie: 45.69%
echo CakeNumber::toPercentage(0.45691, 2, array(
    'multiply' => true
));
```

Introduit dans la version 2.4 : L'argument `$options` avec l'option `multiply` a été ajouté.

`CakeNumber::fromReadableSize`(*string $size*, *$default*)

Paramètres

- **$size** (*string*) – La valeur formatée lisible par un humain.

Cette méthode enlève le format d'un nombre à partir d'une taille de byte lisible par un humain en un nombre entier de bytes. Introduit dans la version 2.3 : Cette méthode a été ajoutée dans 2.3

`CakeNumber::toReadableSize`(*string $dataSize*)

Paramètres

- **$data_size** (*string*) – Le nombre de bytes pour le rendre lisible.

Cette méthode formate les tailles de données dans des formes lisibles pour l'homme. Elle fournit une manière raccourcie de convertir les en KB, MB, GB, et TB. La taille est affichée avec un niveau de précision à deux chiffres, selon la taille de données fournie (ex : les tailles supérieurs sont exprimées dans des termes plus larges) :

```
// appelé avec NumberHelper
echo $this->Number->toReadableSize(0); // 0 Bytes
echo $this->Number->toReadableSize(1024); // 1 KB
echo $this->Number->toReadableSize(1321205.76); // 1.26 MB
echo $this->Number->toReadableSize(5368709120); // 5.00 GB

// appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::toReadableSize(0); // 0 Bytes
echo CakeNumber::toReadableSize(1024); // 1 KB
echo CakeNumber::toReadableSize(1321205.76); // 1.26 MB
echo CakeNumber::toReadableSize(5368709120); // 5.00 GB
```

`CakeNumber::format` (*mixed $number*, *mixed $options=false*)

Cette méthode vous donne beaucoup plus de contrôle sur le formatage des nombres pour l'utilisation dans vos vues (et est utilisée en tant que méthode principale par la plupart des autres méthodes de NumberHelper). L'utilisation de cette méthode pourrait ressembler à cela :

```
// appelé avec NumberHelper
$this->Number->format($number, $options);

// appelé avec CakeNumber
CakeNumber::format($number, $options);
```

Le paramètre $number est le nombre que vous souhaitez formater pour la sortie. Avec aucun $options fourni, le nombre 1236.334 sortirait comme ceci : 1,236. Notez que la précision par défaut est d'aucun chiffre après la virgule.

Le paramètre $options est là où réside la réelle magie de cette méthode.

–Si vous passez un entier alors celui-ci devient le montant de précision pour la fonction.

–Si vous passez un tableau associatif, vous pouvez utiliser les clés suivantes :

–places (integer) : le montant de précision désiré.

–before (string) : à mettre avant le nombre à sortir.

–escape (boolean) : si vous voulez la valeur avant d'être échappée.

–decimals (string) : utilisé pour délimiter les places des décimales dans le nombre.

–thousands (string) : utilisé pour marquer les milliers, millions, ...

Exemple :

```
// appelé avec NumberHelper
echo $this->Number->format('123456.7890', array(
    'places' => 2,
    'before' => '¥ ',
    'escape' => false,
    'decimals' => '.',
    'thousands' => ','
));
// sortie '¥ 123,456.79'
```

```
// appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::format('123456.7890', array(
    'places' => 2,
    'before' => '¥ ',
    'escape' => false,
    'decimals' => '.',
    'thousands' => ','
));
// sortie '¥ 123,456.79'
```

`CakeNumber::formatDelta`(*mixed $number*, *mixed $options=array()*)

Cette méthode affiche les différences en valeur comme un nombre signé :

```
// appelé avec NumberHelper
$this->Number->formatDelta($number, $options);

// appelé avec CakeNumber
CakeNumber::formatDelta($number, $options);
```

Le paramètre $number est le nombre que vous planifiez sur le formatage de sortie. Avec aucun $options fourni, le nombre 1236.334 sortirait 1,236. Notez que la valeur de precision par défaut est aucune décimale.

Le paramètre $options prend les mêmes clés que `CakeNumber::format()` lui-même :
–places (integer) : le montant de precision souhaité.
–before (string) : à mettre avant le nombre sorti.
–after (string) : à mettre après le nombre sorti.
–decimals (string) : utilisé pour délimiter les places de la décimal dans un nombre.
–thousands (string) : utilisé pour marquer les places des centaines, millions, ...
Exemple :

```
// appelé avec NumberHelper
echo $this->Number->formatDelta('123456.7890', array(
    'places' => 2,
    'decimals' => '.',
    'thousands' => ','
));
// sortie '+123,456.79'

// appelé avec CakeNumber
App::uses('CakeNumber', 'Utility');
echo CakeNumber::formatDelta('123456.7890', array(
    'places' => 2,
    'decimals' => '.',
    'thousands' => ','
));
// sortie '+123,456.79'
```

Introduit dans la version 2.3 : Cette méthode a été ajoutée dans 2.3.

CakeTime

class CakeTime

Si vous avez besoin de fonctionnalités `TimeHelper` en-dehors d'une `View`, utilisez la classe `CakeTime` :

```
class UsersController extends AppController {

    public $components = array('Auth');

    public function afterLogin() {
        App::uses('CakeTime', 'Utility');
        if (CakeTime::isToday($this->Auth->user('date_of_birth']))) {
            // greet user with a happy birthday message
            $this->Session->setFlash(__('Happy birthday you...'));
        }
    }
}
```

Introduit dans la version 2.1 : `CakeTime` a été ajouté à partir de `TimeHelper`.

Formatage

`CakeTime::convert`(*$serverTime*, *$timezone = NULL*)

Type retourné integer

Convertit étant donné le time (dans le time zone du serveur) vers le time de l'utilisateur, étant donné son/sa sortie de GMT. :

```
// appel via TimeHelper
echo $this->Time->convert(time(), 'Asia/Jakarta');
// 1321038036

// appel avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::convert(time(), new DateTimeZone('Asia/Jakarta'));
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.

`CakeTime::convertSpecifiers`(*$format*, *$time = NULL*)

Type retourné string

Convertit une chaîne de caractères représentant le format pour la fonction strftime et retourne un format windows safe et i18n aware.

`CakeTime::dayAsSql`(*$dateString*, *$field_name*, *$timezone = NULL*)

Type retourné string

Crée une chaîne de caractères dans le même format que dayAsSql mais nécessite seulement un unique objet date :

```
// Appelé avec TimeHelper
echo $this->Time->dayAsSql('Aug 22, 2011', 'modified');
// (modified >= '2011-08-22 00:00:00') AND (modified <= '2011-08-22 23:59:59')

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::dayAsSql('Aug 22, 2011', 'modified');
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`CakeTime::`**`daysAsSql`**(*$begin*, *$end*, *$fieldName*, *$userOffset = NULL*)

Type retourné string

Retourne une chaîne de caractères dans le format "($field_name >= '2008-01-21 00 :00 :00') AND ($field_name <= '2008-01-25 23 :59 :59')". C'est pratique si vous avez besoin de chercher des enregistrements entre deux dates incluses :

```
// Appelé avec TimeHelper
echo $this->Time->daysAsSql('Aug 22, 2011', 'Aug 25, 2011', 'created');
// (created >= '2011-08-22 00:00:00') AND (created <= '2011-08-25 23:59:59')

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::daysAsSql('Aug 22, 2011', 'Aug 25, 2011', 'created');
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`CakeTime::`**`format`**(*$date*, *$format = NULL*, *$default = false*, *$timezone = NULL*)

Type retourné string

Va retourner une chaîne formatée avec le format donné en utilisant les options de formatage de la fonction PHP date() (http ://www.php.net/manual/en/function.date.php) :

```
// appel via TimeHelper
echo $this->Time->format('F jS, Y h:i A', '2011-08-22 11:53:00');
// August 22nd, 2011 11:53 AM

echo $this->Time->format('r', '+2 days');
// 2 days from now formatted as Sun, 13 Nov 2011 03:36:10 +0800

// appel avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::format('2011-08-22 11:53:00', 'F jS, Y h:i A');
echo CakeTime::format('+2 days', 'r');
```

Vous pouvez aussi fournir la date/time en premier argument. En faisant cela vous devrez utiliser le format `strftime` compatible. Cette signature d'appel vous permet de tirer parti du format de date de la locale ce qui n'est pas possible en utilisant le format de `date()` compatible :

```
// appel avec TimeHelper
echo $this->Time->format('2012-01-13', '%d-%m-%Y', 'invalid');

// appel avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::format('2011-08-22', '%d-%m-%Y');
```

Modifié dans la version 2.2 : Les paramètres `$format` et `$date` sont en ordre opposé par rapport à ce qui se faisait dans 2.1 et suivants. Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants. Le paramètre `$default` remplace le paramètre `$invalid` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$date` accepte aussi maintenant un objet DateTime.

`CakeTime::`**`fromString`**(*$dateString*, *$timezone = NULL*)

Type retourné string

Prend une chaîne et utilise strtotime (http ://us.php.net/manual/en/function.date.php) pour la convertir en une date integer :

```
// Appelé avec TimeHelper
echo $this->Time->fromString('Aug 22, 2011');
// 1313971200

echo $this->Time->fromString('+1 days');
// 1321074066 (+1 day from current date)

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::fromString('Aug 22, 2011');
echo CakeTime::fromString('+1 days');
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`CakeTime::`**`gmt`**(*$dateString = NULL*)

Type retourné integer

Va retourner la date en un nombre défini sur Greenwich Mean Time (GMT). :

```
// Appelé avec TimeHelper
echo $this->Time->gmt('Aug 22, 2011');
// 1313971200

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::gmt('Aug 22, 2011');
```

`CakeTime::`**`i18nFormat`**(*$date*, *$format = NULL*, *$invalid = false*, *$timezone = NULL*)

Type retourné string

Retourne une chaîne de date formatée, étant donné soit un timestamp UNIX soit une chaîne de date valide strtotime(). Il prend en compte le format de la date par défaut pour le langage courant si un fichier LC_TIME est utilisé. Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.

`CakeTime::nice`(*$dateString = NULL, $timezone = NULL, $format = null*)

Type retourné string

Prend une chaîne de date et la sort au format "Tue, Jan 1st 2008, 19 :25" ou avec le param optionnel `$format` :

```
// Appelé avec TimeHelper
echo $this->Time->nice('2011-08-22 11:53:00');
// Mon, Aug 22nd 2011, 11:53

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::nice('2011-08-22 11:53:00');
```

`CakeTime::niceShort`(*$dateString = NULL, $timezone = NULL*)

Type retourné string

Prend une chaîne de date et la sort au format "Jan 1st 2008, 19 :25". Si l'objet date est today, le format sera "Today, 19 :25". Si l'objet date est yesterday, le format sera "Yesterday, 19 :25" :

```
// Appelé avec TimeHelper
echo $this->Time->niceShort('2011-08-22 11:53:00');
// Aug 22nd, 11:53

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::niceShort('2011-08-22 11:53:00');
```

Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`CakeTime::serverOffset`()

Type retourné integer

Retourne la valeur du serveur à partir du GMT dans les secondes.

`CakeTime::timeAgoInWords`(*$dateString, $options = array()*)

Type retourné string

Prendra une chaîne datetime (tout ce qui est parsable par la fonction strtotime() de PHP ou le format de datetime de MySQL) et la convertit en un format de texte comme, "3 weeks, 3 days ago" :

```
// Appelé avec TimeHelper
echo $this->Time->timeAgoInWords('Aug 22, 2011');
// on 22/8/11

echo $this->Time->timeAgoInWords('Aug 22, 2011', array('format' => 'F jS, Y'));
// on August 22nd, 2011

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::timeAgoInWords('Aug 22, 2011');
echo CakeTime::timeAgoInWords('Aug 22, 2011', array('format' => 'F jS, Y'));
```

Utilisez l'option 'end' pour déterminer le point de cutoff pour ne plus utiliser de mots ; par défaut à '+1 month' :

```
// Appelé avec TimeHelper
echo $this->Time->timeAgoInWords('Aug 22, 2011', array('format' => 'F jS, Y', 'end' =>
// On Nov 10th, 2011 it would display: 2 months, 2 weeks, 6 days ago

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
echo CakeTime::timeAgoInWords('Aug 22, 2011', array('format' => 'F jS, Y', 'end' => '+
```

Utilisez l'option 'accuracy' pour déterminer la précision de la sortie. Vous pouvez utiliser ceci pour limiter la sortie :

```
// Si $timestamp est il y a 1 month, 1 week, 5 days et 6 hours
echo CakeTime::timeAgoInWords($timestamp, array(
    'accuracy' => array('month' => 'month'),
    'end' => '1 year'
));
// Sort '1 month ago'
```

Modifié dans la version 2.2 : L'option `accuracy` a été ajoutée.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`CakeTime::toAtom`(*$dateString*, *$timezone = NULL*)

Type retourné string

Va retourner une chaîne de date au format Atom "2008-01-12T00 :00 :00Z" Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`CakeTime::toQuarter`(*$dateString*, *$range = false*)

Type retourné mixed

Va retourner 1, 2, 3 ou 4 dépendant du quart de l'année sur lequel la date tombe. Si range est défini à true, un tableau à deux éléments va être retourné avec les dates de début et de fin au format "2008-03-31" :

```
// Appelé avec TimeHelper
echo $this->Time->toQuarter('Aug 22, 2011');
// Afficherait 3

$arr = $this->Time->toQuarter('Aug 22, 2011', true);
/*
Array
(
    [0] => 2011-07-01
    [1] => 2011-09-30
)
*/

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
```

```
echo CakeTime::toQuarter('Aug 22, 2011');
$arr = CakeTime::toQuarter('Aug 22, 2011', true);
```

Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.Introduit dans la version 2.4 : Les nouveaux paramètres d'option `relativeString` (par défaut à `%s ago`) et `absoluteString` (par défaut à `on %s`) pour permettre la personnalisation de la chaîne de sortie résultante sont maintenant disponibles.

`CakeTime::toRSS`(*$dateString*, *$timezone = NULL*)

Type retourné string

Va retourner une chaîne de date au format RSS "Sat, 12 Jan 2008 00 :00 :00 -0500" Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`CakeTime::toUnix`(*$dateString*, *$timezone = NULL*)

Type retourné integer

Un enrouleur pour fromString. Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime.

`CakeTime::toServer`(*$dateString*, *$timezone = NULL*, *$format = 'Y-m-d H :i :s'*)

Type retourné mixed

Introduit dans la version 2.2 : Retourne une date formatée dans le timezone du serveur.

`CakeTime::timezone`(*$timezone = NULL*)

Type retourné DateTimeZone

Introduit dans la version 2.2 : Retourne un objet timezone à partir d'une chaîne de caractères ou de l'objet timezone de l'utilisateur. Si la fonction est appelée sans paramètres, elle essaie d'obtenir le timezone de la variable de configuration 'Config.timezone'.

`CakeTime::listTimezones`(*$filter = null*, *$country = null*, *$group = true*)

Type retourné array

Introduit dans la version 2.2 : Retourne une liste des identificateurs de timezone.

Tester Time

`CakeTime::isToday`(*$dateString*, *$timezone = NULL*)

`CakeTime::isThisWeek`(*$dateString*, *$timezone = NULL*)

`CakeTime::isThisMonth`(*$dateString*, *$timezone = NULL*)

`CakeTime::isThisYear`(*$dateString*, *$timezone = NULL*)

`CakeTime::wasYesterday`(*$dateString*, *$timezone = NULL*)

`CakeTime::isTomorrow`(*$dateString*, *$timezone = NULL*)

`CakeTime::isFuture`(*$dateString*, *$timezone = NULL*)
Introduit dans la version 2.4.

`CakeTime::isPast`(*$dateString*, *$timezone = NULL*)
Introduit dans la version 2.4.

`CakeTime::wasWithinLast`(*$timeInterval*, *$dateString*, *$timezone = NULL*)
Modifié dans la version 2.2 : Le paramètre `$timezone` remplace le paramètre `$userOffset` utilisé dans 2.1 et suivants.Introduit dans la version 2.2 : Le paramètre `$dateString` accepte aussi maintenant un objet DateTime. Toutes les fonctions ci-dessus retourneront true ou false quand une chaîne de date est passé. `wasWithinLast` prend une option supplémentaire `$time_interval` :

```
// Appelé avec TimeHelper
$this->Time->wasWithinLast($time_interval, $dateString);

// Appelé avec CakeTime
App::uses('CakeTime', 'Utility');
CakeTime::wasWithinLast($time_interval, $dateString);
```

`wasWithinLast` prend un intervalle de time qui est une chaîne au format "3 months" et accepte un intervalle de time en secondes, minutes, heures, jours, semaines, mois et années (pluriels ou non). Si un intervalle de time n'est pas reconnu (par exemple, si il y a une faute de frappe) ensuite ce sera par défaut à days.

Data Sanitization (Nettoyage des Données)

class `Sanitize`

La classe Sanitize de CakePHP peut être utilisée pour rid user-submitted data of malicious data and d'autres informations non souhaitées. Sanitize est une librairie du coeur, donc elle peut être utilisée n'importe où à l'intérieur de votre code, mais est probablement mieux utilisée dans les controllers ou les models.

CakePHP vous protège déjà contre les Injections SQL **si** vous utilisez les méthodes ORM de CakePHP (comme find() et save()) et la notation de tableau appropriée (par ex : array('field' => $value)) au lieu d'un raw SQL. Pour la sanitization contre XSS, il est généralement meilleur de sauvegarder le raw HTML dans la base de données sans modification et sanitize au moment de la sortie/affichage.

Tout ce dont vous avez besoin est d'inclure la librairie du coeur Sanitize (par ex : avant la définition de la classe de controller) :

```
App::uses('Sanitize', 'Utility');

class MyController extends AppController {
    ...
    ...
}
```

Uen fois qu c'est fait, vous pouvez faire des appels statiquement de Sanitize.

static `Sanitize::clean`(*$data*, *$options*)

Paramètres
- **$data** (*mixed*) – Données à nettoyer.

– **$options** (*mixed*) – Options à utiliser pour le nettoyage, voir ci-dessous.

Cette fonction est une force industrielle, multi-purpose cleaner, amenée à être utilisée pour des tableaux entiers (comme $this->data, par exemple). Cette fonction prend un tableau (ou une chaîne) et retourne la version propre. Les opérations de nettoyage suivants sont effectuées sur chaque élément dans le tableau (de façon récursive) :
–Espaces étranges (incluant 0xCA) sont remplacés par des espaces réguliers.
–Double-vérification des chars spéciaux et supprimer les retours chariot pour augmenter la sécurité SQL.
–Ajout de slashes pour SQL (appelle juste la fonction sql soulignée ci-dessus).
–Echange des entrées anti-slash des utilisateurs avec des backslashes.
L'argument $options peut être soit une chaîne, soit un tableau. Quand une chaîne est fournie, c'est le nom de la connection à la base de données. Si un tableau est fourni il sera fusionné avec les options suivantes :
–connection
–odd_spaces
–encode
–dollar
–carriage
–unicode
–escape
–backslash
–remove_html (doit être utilisé en conjonction avec le paramètre encode)
L'utilisation de clean() avec options ressemble à ce qui suit :

```
$this->data = Sanitize::clean($this->data, array('encode' => false));
```

static `Sanitize::escape` (*$string*, *$connection*)

Paramètres
– **$string** (*string*) – Données à nettoyer.
– **$connection** (*string*) – Le nom de la base de données comme nommé dans votre fichier app/Config/database.php.

Utilisé pour échapper les requêtes SQL en ajoutant les slashes, selon la configuration de magic_quotes_gpc su système courant.

static `Sanitize::html` (*$string*, *$options = array()*)

Paramètres
– **$string** (*string*) – Données à nettoyer.
– **$options** (*array*) – Un tableau d'options à utiliser, voir ci-dessous.

Cette méthode prépare des données soumis à l'utilisateur pour l'affichage à l'intérieur du HTML. C'est particulièrement utilise si vous ne voulez pas que les utilisateurs soient capables de casser vos layouts ou d'insérer des images ou scripts à l'intérieur de vos pages HTML. Si l'option $remove est définie à true, le contenu HTML détecté est retiré plutôt que rendu en entités HTML :

```
$badString = '<font size="99" color="#FF0000">HEY</font><script>...</script>';
echo Sanitize::html($badString);
// output: &lt;font size=&quot;99&quot; color=&quot;#FF0000&quot;&gt;HEY&lt;/font&gt;&
echo Sanitize::html($badString, array('remove' => true));
// sort: HEY...
```

Echapper le contenu HTML en utilisant `h` est souvent une meilleur stratégie par rapport à la sanitization, puisqu'elle laisse moins de place aux erreurs, et n'est pas vulnérable aux nouveaux types d'attaques, que la fonction d'échappement ne connait pas.

static `Sanitize::paranoid`(*$string*, *$allowedChars*)

Paramètres

- **$string** (*string*) – Données à nettoyer.
- **$allowedChars** (*string*) – Un tableau de caractères non alpha numériques autorisé.

Cette fonction enlève tout de la cible $string qui n'est pas un caractère plain-jane alphanumeric. La fonction peut être faite pour négliger certains caractères en les faisant passer dans un tableau $allowedChars :

```
$badString = ";:<script><html><   // >@@#";
echo Sanitize::paranoid($badString);
// sort: scripthtml
echo Sanitize::paranoid($badString, array(' ', '@'));
// sort: scripthtml    @@
```

Folder & File

Les utilitaires Folder et File sont des classes pratiques pour aider à la lecture, l'écriture, et l'ajout de fichiers ; Lister les fichiers d'un dossier et autres tâches habituelles liées aux répertoires.

Utilisation basique

Assurez vous que les classes sont chargées en utilisant `App::uses()` :

```
<?php
App::uses('Folder', 'Utility');
App::uses('File', 'Utility');
```

Ensuite nous pouvons configurer une nouvelle instance de dossier :

```
<?php
$dir = new Folder('/path/to/folder');
```

et chercher tous les fichiers *.ctp* à l'intérieur de ce dossier en utilisant les regex :

```
<?php
$files = $dir->find('.*\.ctp');
```

Maintenant nous pouvons boucler sur les fichiers et les lire, écrire ou ajouter aux contenus, ou simplement supprimer le fichier :

```
<?php
foreach ($files as $file) {
    $file = new File($dir->pwd() . DS . $file);
    $contents = $file->read();
    // $file->write('J'écris dans ce fichier');
```

```
    // $file->append('J'ajoute à la fin de ce fichier.');
    // $file->delete(); // Je supprime ce fichier
    $file->close(); // Assurez vous de fermer le fichier quand c'est fini
}
```

API de Folder

class `Folder` (*string $path = false*, *boolean $create = false*, *mixed $mode = false*)

```
<?php
// Crée un nouveau dossier avec les permissions à 0755
$dir = new Folder('/path/to/folder', true, 0755);
```

property `Folder::$path`

Le chemin habituel pour le dossier. `Folder::pwd()` retournera la même information.

property `Folder::$sort`

Dis si la liste des résultats doit être oui ou non rangée par name.

property `Folder::$mode`

Mode à utiliser pour la création de dossiers. par défaut à `0755`. Ne fait rien sur les machines windows.

static `Folder::addPathElement` (*$path*, *$element*)

Type retourné string

Retourne $path avec $element ajouté, avec le bon slash entre-deux :

```
<?php
$path = Folder::addPathElement('/a/path/for', 'testing');
// $path égal /a/path/for/testing
```

`Folder::cd` (*$path*)

Type retourné string

Change le répertoire en $path. Retourne false si échec :

```
<?php
$folder = new Folder('/foo');
echo $folder->path; // Affiche /foo
$folder->cd('/bar');
echo $folder->path; // Affiche /bar
$false = $folder->cd('/non-existent-folder');
```

`Folder::chmod` (*$path*, *$mode = false*, *$recursive = true*, *$exceptions = array ( )*)

Type retourné boolean

Change le mode sur la structure de répertoire de façon récursive. Ceci inclut aussi le changement du mode des fichiers :

```
<?php
$dir = new Folder();
$dir->chmod('/path/to/folder', 0755, true, array('skip_me.php'));
```

`Folder::copy` (*$options = array ( )*)

Type retourné boolean

Copie de façon récursive un répertoire. Le seul paramètre $options peut être soit un chemin à copier soit un tableau d'options :

```
<?php
$folder1 = new Folder('/path/to/folder1');
$folder1->copy('/path/to/folder2');
// Va mettre folder1 et tous son contenu dans folder2

$folder = new Folder('/path/to/folder');
$folder->copy(array(
    'to' => '/path/to/new/folder',
    'from' => '/path/to/copy/from', // va causer un cd() to occur
    'mode' => 0755,
    'skip' => array('skip-me.php', '.git'),
    'scheme' => Folder::SKIP  // Skip directories/files that already exist.
));
```

y a 3 schémas supportés :

–`Folder::SKIP` échapper la copie/déplacement des fichiers & répertoires qui existent dans le répertoire de destination.

–`Folder::MERGE` fusionne les répertoires source/destination. Les fichiers dans le répertoire source vont remplacer les fichiers dans le répertoire de cible. Les contenus du répertoire seront fusionnés.

–`Folder::OVERWRITE` écrase les fichiers & répertoires existant dans la répertoire cible avec ceux dans le répertoire source. Si les deux source et destination contiennent le même sous-répertoire, les contenus du répertoire de cible vont être retirés et remplacés avec celui de la source.

Modifié dans la version 2.3 : La fusion, l'évitement et la surcharge des schémas ont été ajoutés à `copy()`.

static `Folder::correctSlashFor` (*$path*)

Type retourné string

Retourne un ensemble correct de slashes pour un $path donné. (\ pour les chemins Windows et / pour les autres chemins.)

`Folder::create` (*$pathname*, *$mode = false*)

Type retourné boolean

Crée une structure de répertoire de façon récursive. Peut être utilisé pour créer des structures de chemin profond comme */foo/bar/baz/shoe/horn* :

```
<?php
$folder = new Folder();
if ($folder->create('foo' . DS . 'bar' . DS . 'baz' . DS . 'shoe' . DS . 'horn')) {
    // Successfully created the nested folders
}
```

`Folder::delete` (*$path = NULL*)

Type retourné boolean

Efface de façon récursive les répertoires si le système le permet :

```
<?php
$folder = new Folder('foo');
if ($folder->delete()) {
    // Successfully deleted foo its nested folders
}
```

`Folder::dirsize()`

Type retourné integer

Retourne la taille en bytes de ce Dossier et ses contenus.

`Folder::errors()`

Type retourné array

Récupère une erreur de la dernière méthode.

`Folder::find($regexpPattern = '.*', $sort = false)`

Type retourné array

Retourne un tableau de tous les fichiers correspondants dans le répertoire courant :

```
<?php
// Trouve tous les .png dans votre dossier app/webroot/img/ et range les résultats
$dir = new Folder(WWW_ROOT . 'img');
$files = $dir->find('.*\.png', true);
/*
Array
(
    [0] => cake.icon.png
    [1] => test-error-icon.png
    [2] => test-fail-icon.png
    [3] => test-pass-icon.png
    [4] => test-skip-icon.png
)
*/
```

Note : Les méthodes find et findRecursive de folder ne trouvent seulement que des fichiers. Si vous voulez obtenir des dossiers et fichiers, regardez `Folder::read()` ou `Folder::tree()`.

`Folder::findRecursive($pattern = '.*', $sort = false)`

Type retourné array

Retourne un tableau de tous les fichiers correspondants dans et en-dessous du répertoire courant :

```
<?php
// Trouve de façon récursive les fichiers commençant par test ou index
$dir = new Folder(WWW_ROOT);
$files = $dir->findRecursive('(test|index).*');
/*
Array
(
    [0] => /var/www/cake/app/webroot/index.php
    [1] => /var/www/cake/app/webroot/test.php
```

```
        [2] => /var/www/cake/app/webroot/img/test-skip-icon.png
        [3] => /var/www/cake/app/webroot/img/test-fail-icon.png
        [4] => /var/www/cake/app/webroot/img/test-error-icon.png
        [5] => /var/www/cake/app/webroot/img/test-pass-icon.png
    )
    */
```

`Folder::inCakePath`(*$path = ''*)

Type retourné boolean

Retourne true si le Fichier est dans un CakePath donné.

`Folder::inPath`(*$path = '', $reverse = false*)

Type retourné boolean

Retourne true si le Fichier est dans un chemin donné :

```
    <?php
    $Folder = new Folder(WWW_ROOT);
    $result = $Folder->inPath(APP);
    // $result = true, /var/www/example/app/ is in /var/www/example/app/webroot/

    $result = $Folder->inPath(WWW_ROOT . 'img' . DS, true);
    // $result = true, /var/www/example/app/webroot/ est dans /var/www/example/app/webroot
```

static `Folder::isAbsolute`(*$path*)

Type retourné boolean

Retourne true si le $path donné est un chemin absolu.

static `Folder::isSlashTerm`(*$path*)

Type retourné boolean

Retourne true si le $path donné finit par un slash (par exemple. se termine-par-un-slash) :

```
    <?php
    $result = Folder::isSlashTerm('/my/test/path');
    // $result = false
    $result = Folder::isSlashTerm('/my/test/path/');
    // $result = true
```

static `Folder::isWindowsPath`(*$path*)

Type retourné boolean

Retourne true si le $path donné est un chemin Windows.

`Folder::messages`()

Type retourné array

Récupère les messages de la dernière méthode.

`Folder::move`(*$options*)

Type retourné boolean

Déplace le répertoire de façon récursive.

static `Folder::normalizePath($path)`

Type retourné string

Retourne un ensemble correct de slashes pour un $path donné. (\ pour les chemins Windows et / pour les autres chemins.)

`Folder::pwd()`

Type retourné string

Retourne le chemin courant.

`Folder::read($sort = true, $exceptions = false, $fullPath = false)`

Type retourné mixed

Paramètres

- **$sort** (*boolean*) – Si à true, triera les résultats.
- **$exceptions** (*mixed*) – Un tableau de noms de fichiers et de dossiers à ignorer. Si à true ou '.' cette méthode va ignorer les fichiers cachés ou les fichiers commençant par '.'.
- **$fullPath** (*boolean*) – Si à true, va retourner les résultats en utilisant des chemins absolus.

Retourne un tableau du contenu du répertoire courant. Le tableau retourné contient deux tableaux : Un des repertoires et un des fichiers :

```
<?php
$dir = new Folder(WWW_ROOT);
$files = $dir->read(true, array('files', 'index.php'));
/*
Array
(
    [0] => Array
        (
            [0] => css
            [1] => img
            [2] => js
        )
    [1] => Array
        (
            [0] => .htaccess
            [1] => favicon.ico
            [2] => test.php
        )
)
*/
```

`Folder::realpath($path)`

Type retourné string

Récupère le vrai chemin (taking ".." and such into account).

static `Folder::slashTerm($path)`

Type retourné string

Retourne $path avec le slash ajouté à la fin (corrigé pour Windows ou d'autres OS).

`Folder::tree`(*$path = NULL*, *$exceptions = true*, *$type = NULL*)

Type retourné mixed

Retourne un tableau de répertoires imbriqués et de fichiers dans chaque répertoire.

L'API de File

class `File`(*string $path*, *boolean $create = false*, *integer $mode = 493*)

```
<?php
// Crée un nouveau fichier avec les permissions à 0644
$file = new File('/path/to/file.php', true, 0644);
```

property `File::$Folder`

L'objet Folder du fichier.

property `File::$name`

Le nom du fichier avec l'extension. Diffère de `File::name()` qui retourne le nom sans l'extension.

property `File::$info`

Un tableau du fichier info. Utilisez `File::info()` à la place.

property `File::$handle`

Maintient le fichier de gestion des ressources si le fichier est ouvert.

property `File::$lock`

Active le blocage du fichier en lecture et écriture.

property `File::$path`

Le chemin absolu du fichier courant.

`File::append`(*$data*, *$force = false*)

Type retourné boolean

Ajoute une chaîne de caractères $data à ce Fichier.

`File::close`()

Type retourné boolean

Ferme le fichier courant si il est ouvert.

`File::copy`(*$dest*, *$overwrite = true*)

Type retourné boolean

Copie le Fichier vers $dest.

`File::create`()

Type retourné boolean

Crée le Fichier.

`File::delete`()

Type retourné boolean

Supprime le Fichier.

`File::executable()`

Type retourné boolean

Retourne true si le Fichier est executable.

`File::exists()`

Type retourné boolean

Retourne true si le Fichier existe.

`File::ext()`

Type retourné string

Retourne l'extension du Fichier.

`File::Folder()`

Type retourné Folder

Retourne le dossier courant.

`File::group()`

Type retourné integer

Retourne le groupe du Fichier.

`File::info()`

Type retourné string

Retourne l'info du Fichier. Modifié dans la version 2.1 : `File::info()` inclut maintenant les informations filesize & mimetype.

`File::lastAccess()`

Type retourné integer

Retourne le dernier temps d'accès.

`File::lastChange()`

Type retourné integer

Retourne le dernier temps modifié.

`File::md5(`*$maxsize = 5*`)`

Type retourné string

Récupère la md5 Checksum du fichier avec la vérification précédente du Filesize.

`File::name()`

Type retourné string

Retourne le nom du Fichier sans l'extension.

`File::offset(`*$offset = false*, *$seek = 0*`)`

Type retourné mixed

Définit ou récupère l'offset pour le fichier ouvert.

`File::open(`*$mode = 'r'*, *$force = false*`)`

Type retourné boolean

Ouvre le fichier courant avec un $mode donné.

`File::owner()`

Type retourné integer

Retourne le propriétaire du Fichier.

`File::perms()`

Type retourné string

Retourne le "chmod" (permissions) du Fichier.

static `File::prepare`(*$data*, *$forceWindows = false*)

Type retourné string

Prépare une chaîne de caractères ascii pour l'écriture. Convertit les lignes de fin en un terminator correct pour la plateforme courante. Si c'est windows "rn" sera utilisé, toutes les autres plateformes utiliseront "n"

`File::pwd()`

Type retourné string

Retourne un chemin complet du Fichier.

`File::read`(*$bytes = false*, *$mode = 'rb'*, *$force = false*)

Type retourné mixed

Retourne les contenus du Fichier en chaîne de caractère ou retourne false en cas d'échec.

`File::readable()`

Type retourné boolean

Retourne true si le Fichier est lisible.

`File::safe`(*$name = NULL*, *$ext = NULL*)

Type retourné string

Rend le nom de fichier bon pour la sauvegarde.

`File::size()`

Type retourné integer

Retourne le Filesize.

`File::writable()`

Type retourné boolean

Retourne si le Fichier est ouvert en écriture.

`File::write`(*$data*, *$mode = 'w'*, *$force = false*)

Type retourné boolean

Ecrit le $data donné dans le Fichier.

Introduit dans la version 2.1 : `File::mime()`.

`File::mime()`

Type retourné mixed

Récupère le mimetype du Fichier, retourne false en cas d'échec.

HttpSocket

class HttpSocket (*mixed $config = array()*)

CakePHP inclut une classe HttpSocket qui peut être utilisée pour faire des requêtes. C'est un bon moyen pour communiquer avec des services web externes ou des apis distantes.

Faire une requête

Vous pouvez utiliser HttpSocket pour créer la plupart des requêtes HTTP avec les différentes méthodes HTTP.

`HttpSocket::get` (*$uri*, *$query*, *$request*)

Le paramètre `$query` peut soit être une chaîne de caractères, soit un tableau de clés et de valeurs. La méthode get fait une simple requête HTTP GET retournant les résultats :

```
App::uses('HttpSocket', 'Network/Http');

$HttpSocket = new HttpSocket();

// requête chaîne
$results = $HttpSocket->get('http://www.google.com/search', 'q=cakephp');

// requête tableau
$results = $HttpSocket->get('http://www.google.com/search', array('q' => 'cakephp'));
```

`HttpSocket::post` (*$uri*, *$data*, *$request*)

La méthode post fait une simple requête HTTP POST retournant les résultats.

Les paramètres pour la méthode `post` sont presque les mêmes que pour la méthode get, `$uri` est l'adresse web où la requête a été faite ; `$query` est la donnée à poster, que ce soit en chaîne, ou en un tableau de clés et de valeurs :

```
App::uses('HttpSocket', 'Network/Http');

$HttpSocket = new HttpSocket();

// donnée en chaîne
$results = $HttpSocket->post(
    'http://example.com/add',
    'name=test&type=user'
);

// donnée en tableau
$data = array('name' => 'test', 'type' => 'user');
$results = $HttpSocket->post('http://example.com/add', $data);
```

`HttpSocket::put` (*$uri*, *$data*, *$request*)

La méthode put fait une simple requête HTTP PUT retournant les résultats.

Les paramètres pour la méthode `put` est la même que pour la méthode `post()`.

`HttpSocket::delete`(*$uri*, *$query*, *$request*)

La méthode put fait une requête simple HTTP PUT retournant les résultats.

Les paramètres pour la méthode `delete` sont les mêmes que pour la méthode `get()`. Le paramètre `$query` peut soit être une chaîne, soit un tableau d'arguments d'une recherche sous forme de chaîne pour la requête.

`HttpSocket::patch`(*$uri*, *$data*, *$request*)

La méthode patch fait une simple requête HTTP PATCH retournant les résultats.

Les paramètres pour la méthode `patch` sont les mêmes que pour la méthode `post()`. Introduit dans la version 2.4.

`HttpSocket::request`(*$request*)

La méthode request de base qui est appelée à partir de tous les wrappers (get, post, put, delete). Retourne les résultats de la requête.

$request est un tableau à clé avec des options diverses. Voici le format et les configurations par défaut :

```
public $request = array(
    'method' => 'GET',
    'uri' => array(
        'scheme' => 'http',
        'host' => null,
        'port' => 80,
        'user' => null,
        'pass' => null,
        'path' => null,
        'query' => null,
        'fragment' => null
    ),
    'auth' => array(
        'method' => 'Basic',
        'user' => null,
        'pass' => null
    ),
    'version' => '1.1',
    'body' => '',
    'line' => null,
    'header' => array(
        'Connection' => 'close',
        'User-Agent' => 'CakePHP'
    ),
    'raw' => null,
    'redirect' => false,
    'cookies' => array()
);
```

Gérer la réponse

Les réponses des requêtes faîtes avec `HttpSocket` sont des instances de `HttpResponse`. L'objet vous donne quelques méthodes accessor pour accéder au contenu de la réponse HTTP. Cette classe intégre le ArrayAccess (http ://php.net/manual/en/class.arrayaccess.php) et __toString()

(http ://www.php.net/manual/en/language.oop5.magic.php#language.oop5.magic.tostring), donc vous pouvez continuer en utilisant `$http->response` en tableau et le retour des méthodes de requête en chaîne :

```
App::uses('HttpSocket', 'Network/Http');

$http = new HttpSocket();
$response = $http->get('http://www.cakephp.org');

// Check the body for the presence of a title tag.
$titlePos = strpos($response->body, '<title>');

// Récupère le code de statut pour la réponse.
$code = $response->code;
```

`HttpResponse` a les attributs suivants :
- `body` retourne le corps de la réponse HTTP (normalement le HTML).
- `headers` retourne un tableau avec les headers.
- `cookies` retourne un tableau avec les nouveaux cookies (les cookies des autres requêtes ne sont pas stockés ici).
- `httpVersion` retourne une chaîne avec la version de HTTP (à partir de la première ligne dans la réponse).
- `code` retourne l'integer avec le code HTTP.
- `reasonPhrase` retourne la chaîne avec la réponse du code HTTP.
- `raw` retourne la réponse non changée du serveur.

`HttpResponse` expose aussi les méthodes suivantes :
- `body()` retourne le corps.
- `isOk()` retourne si le code est 200 ;
- `isRedirect()` retourne si le code est 301, 302, 303 or 307 et la *localisation* du header est définie.
- `getHeader()` vous permet de récupèrer les headers, voir la prochaine section.

Obtenir des headers à partir d'une réponse Suivant les autres places dans le coeur, HttpSocket ne change pas le cas des headers. **RFC 2616** (http ://tools.ietf.org/html/rfc2616.html) indique que les headers sont insensibles à la casse, et HttpSocket préserve les valeurs que l'hôte distant envoie :

```
HTTP/1.1 200 OK
Date: Mon, 16 Apr 2007 04:14:16 GMT
server: CakeHttp Server
content-tyPe: text/html
```

Votre `$response->headers` (ou `$response['header']`) va contenir les bonnes clés envoyés. Afin d'accéder de manière sécurisé aux champs du header, il est mieux d'utiliser `getHeader()`. Si vos headers ressemblent à ceci :

```
Date: Mon, 16 Apr 2007 04:14:16 GMT
server: CakeHttp Server
content-tyPe: text/html
```

Vous pouvez récupérer les headers ci-dessus en appelant :

```
// $response est une instance de HttpResponse
// récupère le header Content-Type.
$response->getHeader('Content-Type');

// Récupère la date
$response->getHeader('date');
```

Les headers peuvent être récupérés case-insensitively.

Gérer automatiquement une réponse de redirection Quand la réponse a un code de statut de redirection valide (voir `HttpResponse::isRedirect`), une requête supplémentaire peut être automatiquement faîte selon le header *Location* reçu :

```
<?php
App::uses('HttpSocket', 'Network/Http');

$HttpSocket = new HttpSocket();
$response = $HttpSocket->get('http://example.com/redirecting_url', array(), array('redirect
```

L'option *redirect* peut prendre les valeurs suivantes.
- **true** : toutes les réponses de redirection vont entraîner une nouvelle requête conséquente.
- **integer** : La valeur définie est le nombre maximum de redirections autorisées (après l'avoir atteint, la valeur de *redirect* est considerée comme **false**)
- **false** (par défaut) : aucune requête conséquente ne sera fired.

La `$response` retournée sera la dernière, selon les paramètres.

Gérer les certificats SSL Quand vous faites des requêtes vers des services en SSL, HttpSocket va s'attendre à valider le certicat SSL en utilisant la validation peer. Si le certificat échoue la validation peer ou ne correspond pas au nom d'hôte qu'on souhaite accéder, la connection va échouer, et une exception va être lancée. Par défaut HttpSocket va utiliser le fichier d'autorité du certificat mozilla pour vérifier les certificats SSL. Vous pouvez utiliser les options suivantes pour configurer la façon dont les certificats sont gérés :
- **`ssl_verify_peer` Défini à false pour désactiver la vérification SSL.** Ce n'est **pas recommandé**.
- `ssl_verify_host` Défini à false si vous souhaitez ignorer les erreurs de correspondance du nom d'hôte.
- `ssl_allow_self_signed` Défini à true pour activer les certicats que l'on accepte soi-même. Cela nécessite que `ssl_verify_peer` soit activé.
- `ssl_cafile` Défini au chemin absolu du fichier de l'Autorité de Certification que vous souhaitez utiliser pour vérifier les certificats SSL.

Ces options sont fournies dans les arguments du constructeur :

```
$socket = new HttpSocket(array(
    'ssl_allow_self_signed' => true
));
```

Autoriserait les certificats signés soi-même pour toutes les requêtes faîtes avec le socket crée. Introduit dans la version 2.3 : La validation de certificats SSL a été ajoutée dans 2.3.

Créer une classe de réponse personnalisée Vous pouvez créer votre propre classe de réponse pour utiliser HttpSocket. Vous pourriez créer le fichier `app/Lib/Network/Http/YourResponse.php` avec le contenu :

```
App::uses('HttpResponse', 'Network/Http');

class YourResponse extends HttpResponse {

    public function parseResponse($message) {
        parent::parseResponse($message);
        // Make what you want
    }
}
```

Avant votre requête, vous devrez changer la propriété responseClass :

```
App::uses('HttpSocket', 'Network/Http');

$http = new HttpSocket();
$http->responseClass = 'YourResponse';
```

Modifié dans la version 2.3 : Depuis 2.3.0, vous devriez étendre `HttpSocketResponse` à la place. Cela évite un problème commun avec l'extension pecl http.

Télécharger les résultats HttpSocket a une nouvelle méthode appelée *setContentResource()*. En configurant une ressource avec cette méthode, le contenu sera écrit dans la ressource, en utilisant *fwrite()*. Pour télécharger un fichier, vous pouvez faire :

```
App::uses('HttpSocket', 'Network/Http');

$http = new HttpSocket();
$f = fopen(TMP . 'bakery.xml', 'w');
$http->setContentResource($f);
$http->get('http://bakery.cakephp.org/comments.rss');
fclose($f);
```

Note : Les headers ne sont pas inclus dans le fichier, vous récupèrerez seulement le contenu du corps écrit dans votre ressource. Pour désactiver la sauvegarde dans la ressource, utilisez `$http->setContentResource(false)`.

Utiliser l'authentification

HttpSocket supporte des méthodes d'authentification HTTP Basic et Digest. Vous pouvez maintenant créer des objets d'authenrification personnalisée pour supporter des protocoles comme OAuth. Pour utiliser un système d'authentification, vous devez configurer l'instance `HttpSocket` :

```
App::uses('HttpSocket', 'Network/Http');

$http = new HttpSocket();
$http->configAuth('Basic', 'user', 'password');
```

Ce qui est au-dessus configurerait l'instance `HttpSocket` pour utiliser l'authentification Basic en utilisant `user` et `password` en credentials.

Créer un objet d'authentification personnalisé Vous pouvez maintenant créer votre propre méthode d'authentification à utiliser avec HttpSocket. Vous pouvez créer le fichier `app/Lib/Network/Http/YourMethodAuthentication.php` avec le contenu :

```
class YourMethodAuthentication {

/**
 * Authentication
 *
 * @param HttpSocket $http
 * @param array $authInfo
 * @return void
 */
    public static function authentication(HttpSocket $http, &$authInfo) {
        // Faire quelque chose, par exemple définir la valeur $http->request['header']['Au
    }

}
```

Pour configurer HttpSocket afin d'utiliser votre configuraiton auth, vous pouvez utiliser la nouvelle méthode `configAuth()` :

```
$http->configAuth('YourMethod', array('config1' => 'value1', 'config2' => 'value2'));
$http->get('http://secure.your-site.com');
```

La méthode `authentication()` va être appelée pour ajouter aux headers de la requête.

Utiliser un HttpSocket avec un proxy En tant que configuration de auth, vous pouvez configurer une authentification de proxy. Vous pouvez créer votre méthode personnalisée pour authentifier le proxy dans la même classe d'authentification. Par exemple :

```
class YourMethodAuthentication {

/**
 * Authentication
 *
 * @param HttpSocket $http
 * @param array $authInfo
 * @return void
 */
    public static function authentication(HttpSocket $http, &$authInfo) {
        // Faire quelque chose, par exemple définir ma valeur $http->request['header']['Au
    }
```

```
/**
 * Proxy Authentication
 *
 * @param HttpSocket $http
 * @param array $proxyInfo
 * @return void
 */
    public static function proxyAuthentication(HttpSocket $http, &$proxyInfo) {
        // Faire quelque chose, par exemple définir la valeur $http->request['header']['Pro
    }

}
```

Note : Pour utiliser un proxy, vous devez appeler `HttpSocket::configProxy()` semblalble à `HttpSocket::configAuth()`.

Inflector

class `Inflector`

La classe Inflector prend une chaîne de caractères et peut la manipuler pour gérer les variations de mot comme les mises au pluriel ou les mises en Camel et est normalement accessible statiquement. Exemple : `Inflector::pluralize('example')` retourne "examples".

static `Inflector::`**`pluralize`**(*$singular*)
–**Input :** Apple, Orange, Person, Man
–**Output :** Apples, Oranges, People, Men

static `Inflector::`**`singularize`**(*$plural*)
Input : Apples, Oranges, People, Men **Output :** Apple, Orange, Person, Man

static `Inflector::`**`camelize`**(*$underscored*)
–**Input :** Apple_pie, some_thing, people_person
–**Output :** ApplePie, SomeThing, PeoplePerson

static `Inflector::`**`underscore`**(*$camelCase*)
Il doit être noté que les underscores vont seulement convertir les mots formatés en camelCase. Les mots qui contiennent des espaces seront en minuscules, mais ne contiendront pas d'underscore.
–**Input :** applePie, someThing
–**Output :** apple_pie, some_thing

static `Inflector::`**`humanize`**(*$underscored*)
–**Input :** apple_pie, some_thing, people_person
–**Output :** Apple Pie, Some Thing, People Person

static `Inflector::`**`tableize`**(*$camelCase*)
–**Input :** Apple, UserProfileSetting, Person
–**Output :** apples, user_profile_settings, people

static `Inflector::`**`classify`**(*$underscored*)

–**Input :** apples, user_profile_settings, people
–**Output :** Apple, UserProfileSetting, Person

static `Inflector::variable`(*$underscored*)
–**Input :** apples, user_result, people_people
–**Output :** apples, userResult, peoplePeople

static `Inflector::slug`(*$word*, *$replacement* = '_')
Slug convertit les caractères spéciaux en version latins et convertit les caractères ne correspondant pas et les espaces aux underscores. La méthode slug s'attend à un encodage UTF-8.
–**Input :** apple purée
–**Output :** apple_puree

static `Inflector::reset`
Remet l'Inflector à son état initial, utile pour les tests.

static `Inflector::rules`(*$type*, *$rules*, *$reset* = *false*)
Définit de nouvelles règles d'inflection et de translitération à utiliser pour Inflector. Regardez *Configuration de Inflection* pour plus d'informations.

Internationalisation & Localisation

L'une des meilleurs façons pour que votre application ait une audience plus large est de gérer plusieurs langues. Cela peut souvent se révéler être une tâche gigantesque, mais les fonctionnalités d'internationalisation et de localisation dans CakePHP rendront cela plus facile.

D'abord il est important de comprendre quelques terminologies. *Internationalisation* se réfère a la possibilité qu'a une application d'être localisée. Le terme *localisation* se réfère à l'adaptation qu'a une application de répondre aux besoins d'une langue (ou culture) spécifique (par ex : un "locale"). L'internationalisation et la localisation sont souvent abrégées en respectivement i18n et l10n ; 18 et 10 sont le nombre de caractères entre le premier et le dernier caractère.

Internationaliser Votre Application

Il n'y a que quelques étapes à franchir pour passer d'une application mono-langue à une application multi-langue, la première est d'utiliser la fonction `__()` dans votre code. Ci-dessous un exemple d'un code pour une application mono-langue :

```
<h2>Posts</h2>
```

Pour internationaliser votre code, la seule chose à faire est d'entourer la chaîne avec `__()` comme ceci :

```
<h2><?php echo __('Posts') ?></h2>
```

Si vous ne faîtes rien de plus, ces deux bouts de codes donneront un résultat identique - ils renverront le même contenu au navigateur. La fonction `__()` traduira la chaîne passée si une traduction est disponible, sinon elle la renverra non modifiée. Cela fonctionne exactement comme les autres implémentations Gettext Gettext (http ://en.wikipedia.org/wiki/Gettext) (comme les autres fonctions de traductions, comme `__d()` , `__n()` etc).

Après avoir préparé votre code pour le multi-langue, l'étape suivante est de créer votre fichier pot pot file (http ://en.wikipedia.org/wiki/Gettext), qui est le template pour toutes les chaînes traduisibles de votre application. Pour générer votre (vos) fichier(s) pot, tout ce que vous avez à faire est de lancer la tâche i18n *i18n console task* de la console Cake, qui va chercher partout dans votre code où vous avez utilisé une fonction de traduction, et générer le(s) fichier(s) pot pour vous. Vous pouvez (et devez) relancer cette tâche console à chaque fois que vous changez les chaînes traduisibles dans votre code.

Le(s) fichier(s) pot eux mêmes ne sont pas utilisés par CakePHP, ils sont les templates utilisés pour créer ou mettre à jour vos fichiers po, po files (http ://en.wikipedia.org/wiki/Gettext) qui contiennent les traductions. Cake cherchera vos fichiers po dans les dossiers suivants

```
/app/Locale/<locale>/LC_MESSAGES/<domain>.po
```

Le domaine par défaut est 'default', donc votre dossier "locale" devrait ressembler à cela

```
/app/Locale/eng/LC_MESSAGES/default.po (Anglais)
/app/Locale/fra/LC_MESSAGES/default.po (Français)
/app/Locale/por/LC_MESSAGES/default.po (Portugais)
```

Pour créer ou éditer vos fichiers po, il est recommandé de ne pas utiliser votre éditeur de texte préféré. Pour créer un fichier po pour la première fois, il est possible de copier le fichier pot à l'endroit correct et de changer l'extension. *Cependant*, à moins que vous ne soyez familiarisé avec leur format, il est très facile de créer un fichier po invalide, ou de le sauver dans un mauvais encodage de caractères (si vous éditez ces fichiers manuellement, utilisez l'UTF-8 pour éviter les problèmes). Il y a des outils gratuits tel que PoEdit PoEdit (http ://www.poedit.net) qui rendent les tâches d'édition et de mise à jour de vos fichiers po vraiment simples, spécialement pour la mise à jour d'un fichier po existant avec un fichier pot nouvellement mis à jour.

Les codes des locales en trois caractères suivent la norme ISO 639-2 (http ://www.loc.gov/standards/iso639-2/php/code_list.php) mais si vous créez des locales régionales (en_US, en_GB, etc.) Cake les utilisera dans les cas appropriés.

Warning : Dans 2.3 et 2.4, certains codes de langues ont été corrigés pour correspondre au standard ISO. Merci de regarder les guides de migrations correspondants pour plus de détails.

Souvenez-vous que les fichiers po sont utiles pour des messages courts. Si vous pensez que vous aurez à traduire de longs paragraphes, ou des pages entières, vous devriez penser à l'implémentation d'une solution différente. Par ex

```
// Code du fichier App Controller.
public function beforeFilter() {
    $locale = Configure::read('Config.language');
    if ($locale && file_exists(VIEWS . $locale . DS . $this->viewPath)) {
        // utilise /app/views/fra/pages/tos.ctp au lieu de /app/views/pages/tos.ctp
        $this->viewPath = $locale . DS . $this->viewPath;
    }
}
```

ou :

```
// code de la Vue
echo $this->element(Configure::read('Config.language') . '/tos')
```

Internationaliser les plugins CakePHP

Si vous souhaitez inclure des fichiers traduits dans votre application, vous aurez besoin de suivre quelques conventions.

Au lieu de __() et __n() vous devrez uiliser __d() et __dn(). Le D signifie domain. Donc si vous avez un plugin appelé 'DebugKit' vous devrez faire ceci :

```
__d('debug_kit', 'My example text');
```

Utiliser la syntaxe en underscore est important, si vous ne l'utilisez pas, CakePHP ne trouvera pas votre fichier de traduction.

Votre fichier de traduction pour cet exemple devra être dans

```
/app/Plugin/DebugKit/Locale/<locale>/LC_MESSAGES/<domain>.po
```

Et pour les autres langues par rapport à celle par défaut :

```
/app/Plugin/DebugKit/Locale/eng/LC_MESSAGES/debug_kit.po (English)
/app/Plugin/DebugKit/Locale/fra/LC_MESSAGES/debug_kit.po (French)
/app/Plugin/DebugKit/Locale/por/LC_MESSAGES/debug_kit.po (Portuguese)
```

La raison pour cela est que CakePHP va utiliser le nom du plugin en minuscule et avec des underscores, pour le comparer avec le domaine de traduction et va regarder dans le plugin si il y a une correspondance pour le fichier de traduction donné.

Localisation dans CakePHP

Pour changer ou définir le langage de votre application, tout ce que vous avez à faire est dans la partie suivante :

```
Configure::write('Config.language', 'fra');
```

Ceci signale à Cake quelle locale utiliser (si vous utilisez une locale régionale, comme fr_FR, la locale ISO 639-2 (http ://www.loc.gov/standards/iso639-2/php/code_list.php)) sera utilisée au cas ou cela n'existerait pas), vous pouvez changer la langue à n'importe quel moment pendant une requête. Ex : dans votre bootstrap si vous avez défini les paramètres de langue par défaut, dans la partie beforefilter de votre (app) controller si c'est spécifique à la requête ou à l'utilisateur, ou en fait en tout lieu à tout moment avant de passer le message dans une autre langue. Pour définir la langue pour l'utilisateur courant, vous pouvez stocker le paramétrage dans l'objet Session, comme cela :

```
$this->Session->write('Config.language', 'fra');
```

Au début de chacune des requêtes dans la partie `beforeFilter` de votre controller vous devez configurer `Configure` ainsi :

```
class AppController extends Controller{
    public function beforeFilter() {
        if ($this->Session->check('Config.language')) {
            Configure::write('Config.language', $this->Session->read('Config.language'));
        }
    }
}
```

En faisant cela vous assurerez que `I18n` et `TranslateBehavior` accèdent aux même valeurs de langue.

C'est une bonne idée de rendre du contenu public disponible dans plusieurs langues à partir d'une url unique - il deviendra plus facile pour les utilisateurs (et les moteurs de recherches) de trouver ce qu'ils sont venus chercher dans la langue souhaitée. Il y a plusieurs moyens de faire cela, en utilisant un sous domaine de langue spécifique (en.exemple.com,fra.exemple.com, etc.), ou en utilisant un préfixe à l'url comme c'est le cas avec cette application. Vous pourriez également souhaitez glaner l'information depuis l'agent de navigation (browser agent) de l'utilisateur, entre autres choses.

Comme mentionné dans la section précédente, l'affichage des contenus localisés est effectué en utilisant la fonction pratique `__()`, ou une des autres fonctions de traduction qui sont globalement disponibles, mais probablement la plus utilisée dans vos vues. Le premier paramètre de la fonction est utilisé comme le msgid défini dans les fichiers .po.

CakePHP suppose automatiquement que tous les messages d'erreur de validation de votre model dans votre tableau `$validate` sont destinés à être localisées. En exécutant la console i18n ces chaînes seront elles aussi extraites.

Il y a d'autres aspects de localisation de votre application qui ne sont pas couverts par l'utilisation des fonctions de traduction, ce sont les formats date/monnaie. N'oubliez pas que CakePHP est PHP :), donc pour définir les formats de ses éléments vous devez utiliser setlocale (http ://www.php.net/setlocale).

Si vous passez une locale qui n'existe pas sur votre ordinateur setlocale (http ://www.php.net/setlocale) cela n'aura aucun effet. Vous pouvez trouver la liste des locales disponibles en exécutant la commande `locale -a` dans un terminal.

Traduire les erreurs de validation de model

CakePHP va automatiquement extraire l'erreur de validation quand vous utilisez *i18n console task*. Par défaut, le domaine default est utilisé. Ceci peut être surchargé en configurant la propriété `$validationDomain` dans votre model :

```
class User extends AppModel {

    public $validationDomain = 'validation_errors';
}
```

Les paramètres supplémentaires définis dans la règle de validation sont passés à la fonction de traduction. Cela vous permet de créer des messages de validation dynamiques :

```
class User extends AppModel {

    public $validationDomain = 'validation';
```

```
    public $validate = array(
        'username' => array(
                'length' => array(
                'rule' => array('between', 2, 10),
                'message' => 'Username devrait être entre %d et %d caractères'
            )
        )
    )
}
```

Ce qui va faire l'appel interne suivant :

```
__d('validation', 'Username devrait être entre %d et %d caractères', array(2, 10));
```

Journalisation (logging)

Bien que les réglages de la Classe Configure du cœur de CakePHP puissent vraiment vous aider à voir ce qui se passe sous le capot, vous aurez besoin certaines fois de consigner des données sur le disque pour découvrir ce qui se produit. Dans un monde devenu plus dépendant des technologies comme SOAP et AJAX, déboguer peut s'avérer difficile.

Le logging (journalisation) peut aussi être une façon de découvrir ce qui s'est passé dans votre application à chaque instant. Quels termes de recherche ont été utilisés ? Quelles sortes d'erreurs ont été vues par mes utilisateurs ? A quelle fréquence est exécutée une requête particulière ?

La journalisation des données dans CakePHP est facile - la fonction log() est un élément de la classe Object, qui est l'ancêtre commun de la plupart des classes CakePHP. Si le contexte est une classe CakePHP (Model, Controller, Component... n'importe quoi d'autre), vous pouvez loguer (journaliser) vos données. Vous pouvez aussi utiliser `CakeLog::write()` directement. voir *Ecrire dans les logs*

Création et Configuration des flux d'un log (journal)

Les gestionnaires de flux de log peuvent faire partie de votre application, ou partie d'un plugin. Si par exemple vous avez un enregistreur de logs de base de données appelé `DatabaseLog`. Comme faisant parti de votre application il devrait être placé dans `app/Lib/Log/Engine/DatabaseLog.php`. Comme faisant partie d'un plugin il devrait être placé dans `app/Plugin/LoggingPack/Lib/Log/Engine/DatabaseLog.php`. Une fois configuré `CakeLog` va tenter de charger la configuration des flux de logs en appelant''CakeLog : :config()''. La configuration de notre `DatabaseLog` pourrait ressembler à ceci :

```
// pour app/Lib
CakeLog::config('otherFile', array(
    'engine' => 'Database',
    'model' => 'LogEntry',
    // ...
));

// pour un plugin appelé LoggingPack
CakeLog::config('otherFile', array(
```

```
    'engine' => 'LoggingPack.Database',
    'model' => 'LogEntry',
    // ...
));
```

Lorsque vous configurez le flux d'un log le paramètre de `engine` est utilisé pour localiser et charger le handler de log. Toutes les autres propriétés de configuration sont passées au constructeur des flux de log comme un tableau. :

```
App::uses('CakeLogInterface', 'Log');

class DatabaseLog implements CakeLogInterface {
    public function __construct($options = array()) {
        // ...
    }

    public function write($type, $message) {
        // écrire dans la base de données.
    }
}
```

CakePHP n'a pas d'exigences pour les flux de log sinon qu'il doit implémenter une méthode `write`. Cette méthode `write` doit prendre deux paramètres, dans l'ordre `$type, $message`.``$type`` est le type de chaîne du message logué, les valeurs de noyau sont `error`, `warning`, `info` et `debug`. De plus vous pouvez définir vos propres types par leur utilisation en appelant `CakeLog::write`. Introduit dans la version 2.4. Depuis 2.4 le moteur de `FileLog` prend deux nouvelles configurations :

```
- ``size`` Utilisé pour implémenter la rotation de fichier de journal basic.
  Si la taille d'un fichier de log atteint la taille spécifiée, le fichier
  existant est renommé en ajoutant le timestamp au nom du fichier et un
  nouveau fichier de log est crée. Peut être une valeur de bytes en entier
  ou des valeurs de chaînes lisible par l'humain comme '10MB', '100KB' etc.
  Par défaut à 10MB.
- ``rotate`` Les fichiers de log font une rotation à un temps spécifié
  avant d\'être retiré.
  Si la valeur est 0, les versions anciennes seront retirées plutôt que
  mises en rotation. Par défaut à 10.
```

Warning : Avant 2.4 vous deviez inclure le suffixe `Log` dans votre configuration (`LoggingPack.DatabaseLog`). Ce n'est plus nécessaire maintenant. Si vous avez utilisé un moteur de Log comme `'DatabaseLogger` qui ne suit pas la convention d'utiliser un suffixe `Log` pour votre nom de classe, vous devez ajuster votre nom de classe en `DatabaseLog`. Vous devez aussi éviter les noms de classe comme `SomeLogLog` qui inclut le suffixe deux fois à la fin.

Note : Toujours configurer les loggers dans `app/Config/bootstrap.php` Essayer de configurer les loggers ou les loggers de plugin dans core.php provoquera des problèmes, les chemins d'applications n'étant pas encore configurés.

Journalisation des Erreurs et des Exception

Les erreurs et les exception peuvent elles aussi être journalisées. En configurant les valeurs correspondantes dans votre fichier core.php. Les erreurs seront affichées quand debug > 0 et loguées quand debug == 0. Définir `Exception.log` à true pour loguer les exceptions non capturées. Voir *Configuration* pour plus d'information.

Interagir avec les flux de log

Vous pouvez interroger le flux configurés avec `CakeLog::configured()`. Le retour de `configured()` est un tableau de tous les flux actuellement configurés. Vous pouvez rejeter des flux en utilisant `CakeLog::drop()`. Une fois que le flux d'un log à été rejeté il ne recevra plus de messages.

Utilisation de la classe par défaut FileLog

Alors que Cakelog peut être configuré pour écrire à un certain nombre d'adaptateurs de logging (journalisation) configurés par l'utilisateur, il est également livré avec une configuration de logging par défaut qui sera utilisée à chaque fois qu'il n'y a *pas d'autre* adaptateur de logging configuré. Une fois qu'un adaptateur de logging a été configuré vous aurez également à configurer Filelog si vous voulez que le logging de fichier continu.

Comme son nom l'indique FileLog écrit les messages log dans des fichiers. Le type des messages de log en court d'écriture détermine le nom du fichier ou le message sera stocker. Si le type n'est pas fourni, LOG_ERROR est utilisé ce qui à pour effet d'écrire dans le log error. Le chemin par défaut est `app/tmp/logs/$type.log`:

```
// Execute cela dans une classe CakePHP
$this->log("Quelque chose ne fonctionne pas!");

// Aboutit à ce que cela soit ajouté à  app/tmp/logs/error.log
// 2007-11-02 10:22:02 Error: Quelque chose ne fonctionne pas!
```

Vous pouvez spécifier un nom personnalisé en utilisant le premier paramètre. La classe Filelog intégrée par défaut traitera ce nom de log comme le fichier dans lequel vous voulez écrire les logs :

```
// appelé de manière statique
CakeLog::write('activity', 'Un message spécial pour l'activité de logging');

// Aboutit à ce que cela soit ajouté à app/tmp/logs/activity.log (au lieu de error.log)
// 2007-11-02 10:22:02 Activity: Un message spécial pour l'activité de logging
```

Le répertoire configuré doit être accessible en écriture par le serveur web de l'utilisateur pour que la journalisation fonctionne correctement.

Vous pouvez configurer/alterner la localisation de FileLog en utilisant `CakeLog::config()`. FileLog accepte un `chemin` qui permet aux chemins personnalisés d'être utilisés. :

```
CakeLog::config('chemin_perso', array(
    'engine' => 'FileLog',
    'path' => '/chemin/vers/endroit/perso/'
));
```

Logging to Syslog

Introduit dans la version 2.4. Dans les environnements de production, il est fortement recommandé que vous configuriez votre système pour utiliser syslog plutôt que le logger de fichiers. Cela va fonctionner bien mieux que ceux écrits et sera fait (presque) d'une manière non-blocking et le logger de votre système d'exploitation peut être configuré séparément pour faire des rotations de fichier, pré-lancer les écritures ou utiliser un stockage complètement différent pour vos logs.

Utiliser syslog est à peu près comme utiliser le moteur par défaut FileLog, vous devez juste spécifier *SysloLog* comme moteur à utiliser pour la journalisation. Le bout de configuration suivant va remplacer le logger par défaut avec syslog, ceci va être fait dans le fichier *bootstrap.php*.

```
CakeLog::config('default', array(
    'engine' => 'Syslog'
));
```

Le tableau de configuration accepté pour le moteur de journalisation Syslog comprend les clés suivantes :

- *format* : Un template de chaînes sprintf avec deux placeholders, le premier pour le type d'erreur, et le second pour le message lui-même. Cette clé est utile pour ajouter des informations supplémentaires sur le serveur ou la procédure dans le message de log. Par exemple : `%s - Web Server 1 - %s` va ressembler à `error - Web Server 1 - An error occurred in this request` après avoir remplacé les placeholders.
- *prefix* : Une chaine qui va être préfixée à tous les messages de log.
- *flag* : Un drapeau entier utilisé pour l'ouverture de la connection à logger, par défaut *LOG_ODELAY* sera utilisée. Regardez la documentation de *openlog* pour plus d'options.
- *facility* : Le slot de journalisation à utiliser dans syslog. Par défaut *LOG_USER* est utilisé. Regardez la documentation de *syslog* pour plus d'options.

Ecrire dans les logs

Ecrire dans les fichiers peut être réalisé de deux façons. La première est d'utiliser la méthode statique `CakeLog::write()` :

```
CakeLog::write('debug', 'Quelque chose qui ne fonctionne pas');
```

La seconde est d'utiliser la fonction raccourcie log() disponible dans chacune des classes qui étend `Object`. En appelant log() cela appellera en interne CakeLog : :write() :

```
// Exécuter cela dans une classe CakePHP:
$this->log("Quelque chose qui ne fonctionne pas!", 'debug');
```

Tous les flux de log configurés sont écrits séquentiellement à chaque fois que `CakeLog::write()` est appelée. Vous n'avez pas besoin de configurer un flux pour utiliser la journalisation. Si il n'y a pas de flux configuré quand le log est écrit, un flux par `défaut` utilisant la classe de noyau `FileLog` sera configuré pour envoyer en sortie vers `app/tmp/logs/` juste comme CakeLog le faisait dans les précédentes versions.

Scopes de journalisation

Introduit dans la version 2.2. Souvent, vous voudrez configurer différents comportements de journalisation pour différents sous-systèmes ou parties de votre application. Prenez l'exemple d'un magasin e-commerce. Vous voudrez probablement gérer la journalisation pour les commandes et les paiements différemment des autres opérations de journalisation moins critiques.

CakePHP expose ce concept dans les scopes de journalisation. Quand les messages d'erreur sont écrits, vous pouvez inclure un nom scope. Si il y a un logger configuré pour ce scope, les messages de log seront dirigés vers ces loggers. Si un message de log est écrit vers un scope inconnu, les loggers qui gèrent ce niveau de message va journaliser le message. Par exemple :

```
// configurez tmp/logs/shops.log pour recevoir tous les types (niveaux de log), mais seulem
// ceux avec les scope 'orders' et 'payments'
CakeLog::config('shops', array(
    'engine' => 'FileLog',
    'types' => array('warning', 'error'),
    'scopes' => array('orders', 'payments'),
    'file' => 'shops.log',
));

// configurez tmp/logs/payments.log pour recevoir tous les types, mais seulement
// ceux qui ont un scope 'payments'
CakeLog::config('payments', array(
    'engine' => 'SyslogLog',
    'types' => array('info', 'error', 'warning'),
    'scopes' => array('payments')
));

CakeLog::warning('this gets written only to shops stream', 'orders');
CakeLog::warning('this gets written to both shops and payments streams', 'payments');
CakeLog::warning('this gets written to both shops and payments streams', 'unknown');
```

Pour que les scope fonctionnent correctement, vous **devrez** définir les `types` acceptés sur tous les loggers avec lesquels vous voulez utiliser les scopes.

l'API de CakeLog

class `CakeLog`

Une simple classe pour écrire dans les logs (journaux).

static `CakeLog::`**`config`**(*$name*, *$config*)

Paramètres

– **$name** (*string*) – Nom du logger en cours de connexion, utilisé pour rejeter un logger plus tard.
– **$config** (*array*) – Tableau de configuration de l'information et des arguments du constructeur pour le logger.

Connecte un nouveau logger a CakeLog. Chacun des logger connecté reçoit tous les messages de log à chaque fois qu'un message de log est écrit.

static `CakeLog::configured`

Retourne Un tableau des loggers configurés.

Obtient les noms des loggers configurés.

static `CakeLog::drop` (*$name*)

Paramètres
– **$name** (*string*) – Nom du logger duquel vous ne voulez plus recevoir de messages.

static `CakeLog::write` (*$level*, *$message*, *$scope = array()*)

Écrit un message dans tous les loggers configurés. $log indique le type de message créé. $message est le message de l'entrée de log en cours d'écriture. Modifié dans la version 2.2 : `$scope` a été ajouté.

Introduit dans la version 2.2 : Log levels and scopes

static `CakeLog::levels`

Appelle cette méthode sans arguments, ex : `CakeLog::levels()` pour obtenir un niveau de configuration actuel.

Pour ajouter les niveaux supplémentaires 'user0' et 'user1' aux niveaux de log par défaut, utilisez :

```
CakeLog::levels(array('user0', 'user1'));
// ou
CakeLog::levels(array('user0', 'user1'), true);
```

Calling `CakeLog::levels()` va entraîner :

```
array(
    0 => 'emergency',
    1 => 'alert',
    // ...
    8 => 'user0',
    9 => 'user1',
);
```

Pour définir/remplcaer une configuration existante, passez un tableau avec le second argument défini à false :

```
CakeLog::levels(array('user0', 'user1'), false);
```

Calling `CakeLog::levels()` va entraîner :

```
array(
    0 => 'user0',
    1 => 'user1',
);
```

static `CakeLog::defaultLevels`

Retourne Un tableau des valeurs des niveaux de log par défaut.

Efface les niveaux de lof à leurs valeurs originales :

```
array(
    'emergency' => LOG_EMERG,
    'alert'     => LOG_ALERT,
    'critical'  => LOG_CRIT,
    'error'     => LOG_ERR,
    'warning'   => LOG_WARNING,
    'notice'    => LOG_NOTICE,
    'info'      => LOG_INFO,
    'debug'     => LOG_DEBUG,
);
```

static `CakeLog::enabled`(*$streamName*)

Retourne boolean

Vérifie si `$streamName` a été activé.

static `CakeLog::enable`(*$streamName*)

Retourne void

Active le flux `$streamName`.

static `CakeLog::disable`(*$streamName*)

Retourne void

Disable the stream `$streamName`.

static `CakeLog::stream`(*$streamName*)

Retourne Instance de `BaseLog` ou `false` si non retrouvée.

Récupère `$streamName` à partir des flux actifs.

Méthodes pratiques Introduit dans la version 2.2. Les méthodes pratiques suivantes ont été ajoutées au log `$message` avec le niveau de log approprié.

static `CakeLog::emergency`(*$message*, *$scope = array()*)

static `CakeLog::alert`(*$message*, *$scope = array()*)

static `CakeLog::critical`(*$message*, *$scope = array()*)

static `CakeLog::notice`(*$message*, *$scope = array()*)

static `CakeLog::debug`(*$message*, *$scope = array()*)

static `CakeLog::info`(*$message*, *$scope = array()*)

Router

Le Router peut être utilisé pour parser les urls en tableaux contenant les indexes pour le controller, l'action, et tout paramètre, et leur opposé : pour convertir les tableaux url (ex : array('controller' => 'posts', 'action' => 'index')) en chaînes urls.

Lisez en plus sur les façons de *configurer le Router* avec la classe `Router`.

Security

class Security

La librairie security (http ://api.cakephp.org/2.3/class-Security.html) gère les mesures basiques de sécurité comme les méthodes fournies pour le hashage et les données chiffrées.

L'API de Security

static `Security::cipher`(*$text*, *$key*)

Type retourné string

Chiffre/Déchiffre un texte selon la clé donnée :

```
// Chiffre votre mot de passe secret avec my_key
$secret = Security::cipher('hello world', 'my_key');

// Plus tard, déchiffrez votre mot de passe secret
$nosecret = Security::cipher($secret, 'my_key');
```

`cipher()` utilise un cipher XOR **faible** et **ne** doit **pas** être utilisé pour des données importantes ou sensibles.

static `Security::rijndael`(*$text*, *$key*, *$mode*)

Paramètres

- **$text** (*string*) – Le texte à chiffrer.
- **$key** (*string*) – La clé à utiliser pour le chiffrement. Elle doit être plus longue que 32 bytes.
- **$mode** (*string*) – Le mode à utiliser, soit 'encrypt' soit 'decrypt'.

Chiffre/Déchiffre le texte en utilisant le cipher rijndael-256. Ceci nécessite que l'extension mcrypt (http ://php.net/mcrypt) soit installée :

```
// Chiffre quelques données.
$encrypted = Security::rijndael('a secret', Configure::read('Security.key'), 'encrypt'

// Plus tard, le déchiffre.
$decrypted = Security::rijndael($encrypted, Configure::read('Security.key'), 'decrypt'
```

`rijndael()` peut être utilisée pour stocker des données que vous voulez déchiffrer plus tard, comme les contenus des cookies. Il ne devra **jamais** être utilisé pour stocker des mots de passe. Pour cela, vous devrez utiliser la seule méthode de hashage fourni par `hash()` Introduit dans la version 2.2 : `Security::rijndael()` a été ajoutée pour la version 2.2.

static `Security::generateAuthKey`()

Type retourné

string

Génére un hash d'autorisation.

static `Security::getInstance`()

Type retourné object

L'implémentation Singleton pour obtenir l'instance de l'objet.

static `Security::hash`(*$string*, *$type = NULL*, *$salt = false*)

Type retourné string

Crée un hash à partir d'une chaîne en utilisant la méthode donnée. Le Fallback sur la prochaine méthode disponible. Si `$salt` est défini à true, la valeur de salt de l'application sera utilisé :

```
// Utilise la valeur du salt de l'application
$sha1 = Security::hash('CakePHP Framework', 'sha1', true);

// Utilise une valeur du salt personnalisée
$md5 = Security::hash('CakePHP Framework', 'md5', 'my-salt');

// Utilise l'algorithme de hashage par défaut
$hash = Security::hash('CakePHP Framework');
```

`hash()` supporte aussi des algorithms plus sécurisés de hashage comme bcrypt. Quand vous utilisez bcrypt, vous devez vous souvenir de son usage légèrement différent. Créer un hash initial fonctionne de la même façon que les autres algorithmes :

```
// Crée un hash en utilisant bcrypt
Security::setHash('blowfish');
$hash = Security::hash('CakePHP Framework');
```

Au contraire des autres types de hash, la comparaison des valeurs de texte brut devra être faîte comme ce qui suit :

```
// $storedPassword, est un hash bcrypt précédemment généré.
$newHash = Security::hash($newPassword, 'blowfish', $storedPassword);
```

Quand vous comparez les valeurs hashées avec bcrypt, le hash original devra être fourni avec le paramètre `$salt`. Cela permet à bcrypt de réutiliser le même coût et les valeurs de salt, en autorisant le hash généré pour finir avec le même résultat de hash donnant la même valeur d'entrée. Modifié dans la version 2.3 : Le support pour bcrypt a été ajouté dans la version 2.3.

static `Security::inactiveMins`()

Type retourné integer

Pour avoir les minutes d'inactivité autorisées basée sur le niveau de sécurité. :

```
$mins = Security::inactiveMins();
// Si votre config Security.level est défini à 'medium' alors $mins sera égal 100.
```

static `Security::setHash`(*$hash*)

Type retourné void

Définit la méthode de hash par défaut pour l'objet Security. Cela affecte tous les objets en utilisant Security : :hash().

static `Security::validateAuthKey`(*$authKey*)

Type retourné boolean

Valide les hash d'autorisation.

Hash

class `Hash`

Introduit dans la version 2.2. La gestion du tableau, si elle est bien faite, peut être un outil très puissant et utile pour construire du code plus intelligent et plus optimisé. CakePHP offre un ensemble d'utilitaires statiques très utile dans la classe Hash qui vous permet de faire justement cela.

La classe Hash de CakePHP peut être appelée à partir de n'inporte quel model ou controller de la même façon que pour un appel à Inflector Exemple : `Hash::combine()`.

Syntaxe de chemin Hash

La syntaxe de chemin décrite ci-dessous est utilisée par toutes les méthodes dans `Hash`. Les parties de la syntaxe du chemin ne sont pas toutes disponibles dans toutes les méthodes. Une expression en chemin est faite depuis n'importe quel nombre de tokens. Les Tokens sont composés de deux groupes. Les Expressions sont utilisées pour parcourir le tableau de données, alors que les matchers sont utilisés pour qualifier les éléments. Vous appliquez les matchers aux élements de l'expression.

Expression	Définition
`{n}`	Représente une clé numérique. Va matcher toute chaîne ou clé numérique.
`{s}`	Représente une chaîne. Va matcher toute valeur de chaîne y compris les valeurs de chaîne numérique.
`Foo`	Matche les clés avec exactement la même valeur keys with the exact same value.

Tous les éléments d'expression supportent toutes les méthodes. En plus des éléments d'expression, vous pouvez utiliser les attributs qui matchent avec les méthodes comme `extract()`.

Matcher	Definition
`[id]`	Match les éléments avec une clé de tableau donnée.
`[id=2]`	Match les éléments avec un id égal à 2.
`[id!=2]`	Match les éléments avec un id non égal à 2.
`[id>2]`	Match les éléments avec un id supérieur à 2.
`[id>=2]`	Match les éléments avec un id supérieur égal à 2.
`[id<2]`	Match les éléments avec un id inférieur à 2.
`[id<=2]`	Match les éléménts avec un id inférieur ou égal à 2.
`[text=/.../]`	Match les éléments qui ont des valeurs matchant avec l'expression régulière à l'intérieur de `....`

static `Hash::get` (*array $data*, *$path*)

Type retourné mixed

`get()` est une version simplifiée de `extract()`, elle ne supporte que les expressions de chemin direct. Les chemins avec `{n}`, `{s}` ou les matchers ne sont pas supportés. Utilisez `get()` quand vous voulez exactement une valeur sortie d'un tableau.

static `Hash::extract` (*array $data*, *$path*)

Type retourné array

`Hash::extract()` supporte toutes les expressions, les components matcher de la *Syntaxe de chemin Hash*. Vous pouvez utilisez l'extract pour récupèrer les données à partir des tableaux, le long des chemins arbitraires rapidement sans avoir à parcourir les structures de données. A la place, vous utilisez les expressions de chemin pour qualifier les éléments que vous souhaitez retourner

```
// Utilisation habituelle:
$users = $this->User->find("all");
$results = Hash::extract($users, '{n}.User.id');
// $results égal à:
// array(1,2,3,4,5,...);
```

static `Hash::insert`(*array $data*, *$path*, *$values = null*)

Type retourné array

Insère $data dans un tableau comme défini par $path. Cette méthode supporte seulement les types d'expression de *Syntaxe de chemin Hash* :

```
$a = array(
    'pages' => array('name' => 'page')
);
$result = Hash::insert($a, 'files', array('name' => 'files'));
// $result ressemble maintenant à:
Array
(
    [pages] => Array
        (
            [name] => page
        )
    [files] => Array
        (
            [name] => files
        )
)
```

Vous pouvez utiliser les chemins en utilisant `{n}` et `{s}` pour insérer des données dans des points multiples :

```
$users = $this->User->find('all');
$users = Set::insert($users, '{n}.User.new', 'value');
```

static `Hash::remove`(*array $data*, *$path = null*)

Type retourné array

Retire tous les éléments d'un tableau qui matche avec $path. Cette méthode supporte tous les éléments d'expression de *Syntaxe de chemin Hash* :

```
$a = array(
    'pages' => array('name' => 'page'),
    'files' => array('name' => 'files')
);
$result = Hash::remove($a, 'files');
/* $result ressemble maintenant à:
    Array
    (
```

```
            [pages] => Array
                (
                    [name] => page
                )

        )
    */
```

L'utilisation de `{n}` et `{s}` vous autorisera à retirer les valeurs multiples en une fois.

static `Hash::combine` (*array $data*, *$keyPath = null*, *$valuePath = null*, *$groupPath = null*)

Type retourné array

Crée un tableau associatif en utilisant $keyPath en clé pour le chemin à construire, et optionnellement $valuePath comme chemin pour récupèrer les valeurs. Si $valuePath n'est pas spécifiée, ou ne matche rien, les valeurs seront initialisées à null. Vous pouvez grouper en option les valeurs par ce qui est obtenu en suivant le chemin spécifié dans $groupPath. :

```
    $a = array(
        array(
            'User' => array(
                'id' => 2,
                'group_id' => 1,
                'Data' => array(
                    'user' => 'mariano.iglesias',
                    'name' => 'Mariano Iglesias'
                )
            )
        ),
        array(
            'User' => array(
                'id' => 14,
                'group_id' => 2,
                'Data' => array(
                    'user' => 'phpnut',
                    'name' => 'Larry E. Masters'
                )
            )
        ),
    );

    $result = Hash::combine($a, '{n}.User.id');
    /* $result ressemble maintenant à:
        Array
        (
            [2] =>
            [14] =>
        )
    */

    $result = Hash::combine($a, '{n}.User.id', '{n}.User.Data');
    /* $result ressemble maintenant à:
        Array
        (
```

```
            [2] => Array
                (
                    [user] => mariano.iglesias
                    [name] => Mariano Iglesias
                )
            [14] => Array
                (
                    [user] => phpnut
                    [name] => Larry E. Masters
                )
        )
    */

    $result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.name');
    /* $result ressemble maintenant à:
        Array
        (
            [2] => Mariano Iglesias
            [14] => Larry E. Masters
        )
    */

    $result = Hash::combine($a, '{n}.User.id', '{n}.User.Data', '{n}.User.group_id');
    /* $result ressemble maintenant à:
        Array
        (
            [1] => Array
                (
                    [2] => Array
                        (
                            [user] => mariano.iglesias
                            [name] => Mariano Iglesias
                        )
                )
            [2] => Array
                (
                    [14] => Array
                        (
                            [user] => phpnut
                            [name] => Larry E. Masters
                        )
                )
        )
    */

    $result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.name', '{n}.User.group_id');
    /* $result ressemble maintenant à:
        Array
        (
            [1] => Array
                (
                    [2] => Mariano Iglesias
                )
```

```
            [2] => Array
                (
                    [14] => Larry E. Masters
                )
        )
    */
```

Vous pouvez fournir des tableaux pour les deux $keyPath et $valuePath. Si vous le faîtes, la première valeur sera utlisée comme un format de chaîne de caractères, pour les valeurs extraites par les autres chemins :

```
$result = Hash::combine(
    $a,
    '{n}.User.id',
    array('%s: %s', '{n}.User.Data.user', '{n}.User.Data.name'),
    '{n}.User.group_id'
);
/* $result ressemble maintenant à:
    Array
    (
        [1] => Array
            (
                [2] => mariano.iglesias: Mariano Iglesias
            )
        [2] => Array
            (
                [14] => phpnut: Larry E. Masters
            )
    )
*/

$result = Hash::combine(
    $a,
    array('%s: %s', '{n}.User.Data.user', '{n}.User.Data.name'),
    '{n}.User.id'
);
/* $result ressemble maintenant à:
    Array
    (
        [mariano.iglesias: Mariano Iglesias] => 2
        [phpnut: Larry E. Masters] => 14
    )
*/
```

static `Hash::format` (*array $data*, *array $paths*, *$format*)

Type retourné array

Retourne une série de valeurs extraites d'un tableau, formaté avec un format de chaîne de caractères :

```
$data = array(
    array(
        'Person' => array(
            'first_name' => 'Nate',
            'last_name' => 'Abele',
```

```
                'city' => 'Boston',
                'state' => 'MA',
                'something' => '42'
            )
        ),
        array(
            'Person' => array(
                'first_name' => 'Larry',
                'last_name' => 'Masters',
                'city' => 'Boondock',
                'state' => 'TN',
                'something' => '{0}'
            )
        ),
        array(
            'Person' => array(
                'first_name' => 'Garrett',
                'last_name' => 'Woodworth',
                'city' => 'Venice Beach',
                'state' => 'CA',
                'something' => '{1}'
            )
        )
    );

    $res = Hash::format($data, array('{n}.Person.first_name', '{n}.Person.something'), '%2
    /*
    Array
    (
        [0] => 42, Nate
        [1] => 0, Larry
        [2] => 0, Garrett
    )
    */

    $res = Hash::format($data, array('{n}.Person.first_name', '{n}.Person.something'), '%1
    /*
    Array
    (
        [0] => Nate, 42
        [1] => Larry, 0
        [2] => Garrett, 0
    )
    */
```

static `Hash::contains`(*array $data*, *array $needle*)

Type retourné boolean

Détermine si un Hash ou un tableau contient les clés et valeurs exactes d'un autre :

```
    $a = array(
        0 => array('name' => 'main'),
        1 => array('name' => 'about')
    );
```

```
$b = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about'),
    2 => array('name' => 'contact'),
    'a' => 'b'
);

$result = Hash::contains($a, $a);
// true
$result = Hash::contains($a, $b);
// false
$result = Hash::contains($b, $a);
// true
```

static `Hash::check` (*array $data*, *string $path = null*)

rtype boolean

Vérifie si un chemin particulier est défini dans un tableau :

```
$set = array(
    'My Index 1' => array('First' => 'The first item')
);
$result = Hash::check($set, 'My Index 1.First');
// $result == True

$result = Hash::check($set, 'My Index 1');
// $result == True

$set = array(
    'My Index 1' => array('First' =>
        array('Second' =>
            array('Third' =>
                array('Fourth' => 'Heavy. Nesting.'))))
);
$result = Hash::check($set, 'My Index 1.First.Second');
// $result == True

$result = Hash::check($set, 'My Index 1.First.Second.Third');
// $result == True

$result = Hash::check($set, 'My Index 1.First.Second.Third.Fourth');
// $result == True

$result = Hash::check($set, 'My Index 1.First.Seconds.Third.Fourth');
// $result == False
```

static `Hash::filter` (*array $data*, *$callback = array('Hash', 'filter')*)

Type retourné array

Filtre les éléments vides en dehors du tableau, en excluant '0'. Vous pouvez aussi fournir un $callback personnalisé pour filtrer les éléments de tableau. Votre callback devrait retourner `false` pour retirer les éléments du tableau résultant :

```
$data = array(
    '0',
    false,
    true,
    0,
    array('one thing', 'I can tell you', 'is you got to be', false)
);
$res = Hash::filter($data);

/* $data ressemble maintenant à:
    Array (
        [0] => 0
        [2] => true
        [3] => 0
        [4] => Array
            (
                [0] => one thing
                [1] => I can tell you
                [2] => is you got to be
            )
    )
*/
```

static `Hash::flatten`(*array $data*, *string $separator* = '.')

Type retourné array

Réduit un tableau multi-dimensionnel en un tableau à une seule dimension :

```
$arr = array(
    array(
        'Post' => array('id' => '1', 'title' => 'First Post'),
        'Author' => array('id' => '1', 'user' => 'Kyle'),
    ),
    array(
        'Post' => array('id' => '2', 'title' => 'Second Post'),
        'Author' => array('id' => '3', 'user' => 'Crystal'),
    ),
);
$res = Hash::flatten($arr);
/* $res ressemble maintenant à:
    Array (
        [0.Post.id] => 1
        [0.Post.title] => First Post
        [0.Author.id] => 1
        [0.Author.user] => Kyle
        [1.Post.id] => 2
        [1.Post.title] => Second Post
        [1.Author.id] => 3
        [1.Author.user] => Crystal
    )
*/
```

static `Hash::expand`(*array $data*, *string $separator* = '.')

Type retourné array

Développe un tableau qui a déjà été aplatie avec `Hash::flatten()` :

```
$data = array(
    '0.Post.id' => 1,
    '0.Post.title' => First Post,
    '0.Author.id' => 1,
    '0.Author.user' => Kyle,
    '1.Post.id' => 2,
    '1.Post.title' => Second Post,
    '1.Author.id' => 3,
    '1.Author.user' => Crystal,
);
$res = Hash::expand($data);
/* $res ressemble maintenant à:
array(
    array(
        'Post' => array('id' => '1', 'title' => 'First Post'),
        'Author' => array('id' => '1', 'user' => 'Kyle'),
    ),
    array(
        'Post' => array('id' => '2', 'title' => 'Second Post'),
        'Author' => array('id' => '3', 'user' => 'Crystal'),
    ),
);
*/
```

static `Hash::merge`(*array $data*, *array $merge*[, *array $n*])

Type retourné array

Cette fonction peut être vue comme un hybride entre le `array_merge` et le `array_merge_recursive` de PHP. La différence entre les deux est que si une clé du tableau contient un autre tableau, alors la fonction se comporte de façon récursive (pas comme `array_merge`) mais ne le fait pas pour les clés contenant les chaînes de caractères (pas comme `array_merge_recursive`).

Note : Cette fonction va fonctionner avec un montant illimité d'arguments et convertit les paramètres de non-tableau en tableaux.

```
$array = array(
    array(
        'id' => '48c2570e-dfa8-4c32-a35e-0d71cbdd56cb',
        'name' => 'mysql raleigh-workshop-08 < 2008-09-05.sql ',
        'description' => 'Importing an sql dump'
    ),
    array(
        'id' => '48c257a8-cf7c-4af2-ac2f-114ecbdd56cb',
        'name' => 'pbpaste | grep -i Unpaid | pbcopy',
        'description' => 'Remove all lines that say "Unpaid".',
    )
);
$arrayB = 4;
```

```
$arrayC = array(0 => "test array", "cats" => "dogs", "people" => 1267);
$arrayD = array("cats" => "felines", "dog" => "angry");
$res = Hash::merge($array, $arrayB, $arrayC, $arrayD);

/* $res ressemble maintenant à:
Array
(
    [0] => Array
        (
            [id] => 48c2570e-dfa8-4c32-a35e-0d71cbdd56cb
            [name] => mysql raleigh-workshop-08 < 2008-09-05.sql
            [description] => Importing an sql dump
        )
    [1] => Array
        (
            [id] => 48c257a8-cf7c-4af2-ac2f-114ecbdd56cb
            [name] => pbpaste | grep -i Unpaid | pbcopy
            [description] => Remove all lines that say "Unpaid".
        )
    [2] => 4
    [3] => test array
    [cats] => felines
    [people] => 1267
    [dog] => angry
)
*/
```

static `Hash::numeric` (*array $data*)

Type retourné boolean

Vérifie pour voir si toutes les valeurs dans le tableau sont numériques :

```
$data = array('one');
$res = Hash::numeric(array_keys($data));
// $res est à true

$data = array(1 => 'one');
$res = Hash::numeric($data);
// $res est à false
```

static `Hash::dimensions` (*array $data*)

Type retourné integer

Compte les dimensions d'un tableau. Cette méthode va seulement considérer la dimension du premier élément dans le tableau :

```
$data = array('one', '2', 'three');
$result = Hash::dimensions($data);
// $result == 1

$data = array('1' => '1.1', '2', '3');
$result = Hash::dimensions($data);
// $result == 1
```

```
$data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => '3.1.1'));
$result = Hash::dimensions($data);
// $result == 2

$data = array('1' => '1.1', '2', '3' => array('3.1' => '3.1.1'));
$result = Hash::dimensions($data);
// $result == 1


$data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => array('3.1.1'
$result = Hash::countDim($data);
// $result == 2
```

static `Hash::maxDimensions`(*array $data*)

Similaire à `dimensions()`, cependant cette méthode retourne le nombre le plus profond de dimensions de tout élément dans le tableau :

```
$data = array('1' => '1.1', '2', '3' => array('3.1' => '3.1.1'));
$result = Hash::dimensions($data, true);
// $result == 2

$data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => array('3.1.1'
$result = Hash::countDim($data, true);
// $result == 3
```

static `Hash::map`(*array $data*, *$path*, *$function*)

Crée un nouveau tableau, en extrayant $path, et mappe $function à travers les résultats. Vous pouvez utiliser les deux, expression et le matching d'éléments avec cette méthode.

static `Hash::reduce`(*array $data*, *$path*, *$function*)

Crée une valeur unique, en extrayant $path, et en réduisant les résultats extraits avec $function. Vous pouvez utiliser les deux, expression et le matching d'éléments avec cette méthode.

static `Hash::sort`(*array $data*, *$path*, *$dir*, *$type = 'regular'*)

Type retourné array

Trie un tableau selon n'importe quelle valeur, déterminé par une *Syntaxe de chemin Hash*. Seuls les élements de type expression sont supportés par cette méthode :

```
$a = array(
    0 => array('Person' => array('name' => 'Jeff')),
    1 => array('Shirt' => array('color' => 'black'))
);
$result = Hash::sort($a, '{n}.Person.name', 'asc');
/* $result ressemble maintenant à:
    Array
    (
        [0] => Array
            (
                [Shirt] => Array
                    (
                        [color] => black
                    )
```

```
                )
            [1] => Array
                (
                    [Person] => Array
                        (
                            [name] => Jeff
                        )
                )
        )
    */
```

`$dir` peut être soit `asc`, soit `desc`. Le ``$type` peut être une des valeurs suivantes :

–`regular` pour le trier régulier.

–`numeric` pour le tri des valeurs avec leurs valeurs numériques équivalentes.

–`string` pour le tri des valeurs avec leur valeur de chaîne.

–`natural` pour trier les valeurs d'une façon humaine. Va trier `foo10` en-dessous de `foo2` par exemple. Le tri naturel a besoin de PHP 5.4 ou supérieur.

static `Hash::diff`(*array $data*, *array $compare*)

Type retourné array

Calcule la différence entre deux tableaux :

```
$a = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about')
);
$b = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about'),
    2 => array('name' => 'contact')
);

$result = Hash::diff($a, $b);
/* $result ressemble maintenant à:
    Array
    (
        [2] => Array
            (
                [name] => contact
            )
    )
*/
```

static `Hash::mergeDiff`(*array $data*, *array $compare*)

Type retourné array

Cette fonction fusionne les deux tableaux et pousse les différences dans les données à la fin du tableau résultant.

Exemple 1

```
$array1 = array('ModelOne' => array('id' => 1001, 'field_one' => 'a1.m1.f1', 'field_tw
$array2 = array('ModelOne' => array('id' => 1003, 'field_one' => 'a3.m1.f1', 'field_tw
```

```
$res = Hash::mergeDiff($array1, $array2);

/* $res ressemble maintenant à:
    Array
    (
        [ModelOne] => Array
            (
                [id] => 1001
                [field_one] => a1.m1.f1
                [field_two] => a1.m1.f2
                [field_three] => a3.m1.f3
            )
    )
*/
```

Exemple 2

```
$array1 = array("a" => "b", 1 => 20938, "c" => "string");
$array2 = array("b" => "b", 3 => 238, "c" => "string", array("extra_field"));
$res = Hash::mergeDiff($array1, $array2);
/* $res ressemble maintenant à:
    Array
    (
        [a] => b
        [1] => 20938
        [c] => string
        [b] => b
        [3] => 238
        [4] => Array
            (
                [0] => extra_field
            )
    )
*/
```

static `Hash::normalize`(*array $data*, *$assoc = true*)

Type retourné array

Normalise un tableau. Si `$assoc` est à true, le tableau résultant sera normalisé en un tableau associatif. Les clés numériques avec les valeurs, seront convertis en clés de type chaîne avec des valeurs null. Normaliser un tableau, facilite l'utilisation des résultats avec `Hash::merge()` :

```
$a = array('Tree', 'CounterCache',
    'Upload' => array(
        'folder' => 'products',
        'fields' => array('image_1_id', 'image_2_id')
    )
);
$result = Hash::normalize($a);
/* $result ressemble maintenant à:
    Array
    (
        [Tree] => null
        [CounterCache] => null
```

```
            [Upload] => Array
                (
                    [folder] => products
                    [fields] => Array
                        (
                            [0] => image_1_id
                            [1] => image_2_id
                        )
                )
        )
    */

    $b = array(
        'Cacheable' => array('enabled' => false),
        'Limit',
        'Bindable',
        'Validator',
        'Transactional'
    );
    $result = Hash::normalize($b);
    /* $result ressemble maintenant à:
        Array
        (
            [Cacheable] => Array
                (
                    [enabled] => false
                )

            [Limit] => null
            [Bindable] => null
            [Validator] => null
            [Transactional] => null
        )
    */
```

static `Hash::nest` (*array $data*, *array $options = array()*)

Prend un ensemble de tableau aplati, et crée une structure de données imbriquée ou chaînée. Utilisé par des méthodes comme `Model::find('threaded')`.

Options :

–`children` Le nom de la clé à utiliser dans l'ensemble de résultat pour les enfants. Par défaut à 'children'.

–`idPath` Le chemin vers une clé qui identifie chaque entrée. Doit être compatible avec `Hash::extract()`. Par défaut à `{n}.$alias.id`

–`parentPath` Le chemin vers une clé qui identifie le parent de chaque entrée. Doit être compatible avec `Hash::extract()`. Par défaut à `{n}.$alias.parent_id`.

–`root` L'id du résultat le plus désiré.

Exemple :

```
    $data = array(
        array('ModelName' => array('id' => 1, 'parent_id' => null)),
        array('ModelName' => array('id' => 2, 'parent_id' => 1)),
        array('ModelName' => array('id' => 3, 'parent_id' => 1)),
```

```
        array('ModelName' => array('id' => 4, 'parent_id' => 1)),
        array('ModelName' => array('id' => 5, 'parent_id' => 1)),
        array('ModelName' => array('id' => 6, 'parent_id' => null)),
        array('ModelName' => array('id' => 7, 'parent_id' => 6)),
        array('ModelName' => array('id' => 8, 'parent_id' => 6)),
        array('ModelName' => array('id' => 9, 'parent_id' => 6)),
        array('ModelName' => array('id' => 10, 'parent_id' => 6))
    );

    $result = Hash::nest($data, array('root' => 6));
    /* $result ressemble maintenant à:
    array(
            (int) 0 => array(
                'ModelName' => array(
                    'id' => (int) 6,
                    'parent_id' => null
                ),
                'children' => array(
                    (int) 0 => array(
                        'ModelName' => array(
                            'id' => (int) 7,
                            'parent_id' => (int) 6
                        ),
                        'children' => array()
                    ),
                    (int) 1 => array(
                        'ModelName' => array(
                            'id' => (int) 8,
                            'parent_id' => (int) 6
                        ),
                        'children' => array()
                    ),
                    (int) 2 => array(
                        'ModelName' => array(
                            'id' => (int) 9,
                            'parent_id' => (int) 6
                        ),
                        'children' => array()
                    ),
                    (int) 3 => array(
                        'ModelName' => array(
                            'id' => (int) 10,
                            'parent_id' => (int) 6
                        ),
                        'children' => array()
                    )
                )
            )
        )
        */
```

Set

class Set

La gestion de tableau, si elle est bien faite, peut être un outil très puissant et utile pour construire plus malin, et du code plus optimisé. CakePHP offre un ensemble d'utilitaires statiques très utile dans la classe Set qui vous permet justement de faire cela.

La classe Set de CakePHP peut être appelée par n'importe quel model ou controller de la même façon que l'Inflector est appelé. Exemple : `Set::combine()`. Obsolète depuis la version 2.2 : La classe Set a été dépréciée dans 2.2 en faveur de la classe `Hash`. Il offre une interface et une API plus cohérente.

La syntaxe du Chemin Set-compatible

La syntaxe de Chemin est utilisée par sorte (par exemple), et est utilisée pour définir un chemin.

Exemple d'utilisation (en utilisant `Set::sort()`) :

```
$a = array(
    0 => array('Person' => array('name' => 'Jeff'), 'Friend' => array(array('name' => 'Nate
    1 => array('Person' => array('name' => 'Tracy'),'Friend' => array(array('name' => 'Linc
    2 => array('Person' => array('name' => 'Adam'),'Friend' => array(array('name' => 'Bob')
);
$result = Set::sort($a, '{n}.Person.name', 'asc');
/* result now looks like
 array(
    0 => array('Person' => array('name' => 'Adam'),'Friend' => array(array('name' => 'Bob',
    1 => array('Person' => array('name' => 'Jeff'), 'Friend' => array(array('name' => 'Nate
    2 => array('Person' => array('name' => 'Tracy'),'Friend' => array(array('name' => 'Linc
);
*/
```

Comme vous pouvez le voir dans l'exemple ci-dessus, certaines choses sont entourées de {}, d'autres non. Dans la table ci-dessous, vous pouvez voir quelles options sont disponibles.

Expression	Definition
{n}	Représente une clé numérique
{s}	Représente une chaîne
Foo	Toute chaîne (sans les accolades fermantes) est traitée comme une chaîne littérale.
{[a-z]+}	Toute chaîne entre accolades (à part {n} et {s}) est interpretée comme une expression régulière.

static `Set::apply`(*$path*, *$array*, *$callback*, *$options = array()*)

Type retourné mixed

Appliquer un callback aux éléments d'un tableau extrait par un chemin Set : :extract compatible :

```
    $data = array(
        array('Movie' => array('id' => 1, 'title' => 'movie 3', 'rating' => 5)),
        array('Movie' => array('id' => 1, 'title' => 'movie 1', 'rating' => 1)),
        array('Movie' => array('id' => 1, 'title' => 'movie 2', 'rating' => 3)),
```

```
);

$result = Set::apply('/Movie/rating', $data, 'array_sum');
// résultat égal à 9

$result = Set::apply('/Movie/title', $data, 'strtoupper', array('type' => 'map'));
// résultat égal à array('MOVIE 3', 'MOVIE 1', 'MOVIE 2')
// $options sont: - type : peut être 'pass' uses call_user_func_array(), 'map' uses ar
```

static `Set::check` (*$data*, *$path = null*)

Type retourné boolean/array

Vérifie si un chemin particulier est défini dans un tableau. Si $path est vide, $data va être retournée au lieu d'une valeur boléenne :

```
$set = array(
    'My Index 1' => array('First' => 'The first item')
);
$result = Set::check($set, 'My Index 1.First');
// $result == True
$result = Set::check($set, 'My Index 1');
// $result == True
$result = Set::check($set, array());
// $result == array('My Index 1' => array('First' => 'The first item'))
$set = array(
    'My Index 1' => array('First' =>
        array('Second' =>
            array('Third' =>
                array('Fourth' => 'Heavy. Nesting.'))))
);
$result = Set::check($set, 'My Index 1.First.Second');
// $result == True
$result = Set::check($set, 'My Index 1.First.Second.Third');
// $result == True
$result = Set::check($set, 'My Index 1.First.Second.Third.Fourth');
// $result == True
$result = Set::check($set, 'My Index 1.First.Seconds.Third.Fourth');
// $result == False
```

static `Set::classicExtract` (*$data*, *$path = null*)

Type retourné array

Récupère une valeur d'un tableau ou d'un objet qui est contenu dans un chemin donné en utilisant un tableau en une syntaxe de tableau, par ex :

–"{n}.Person.{[a-z]+}" - "{n}" représente une clé numérique, "Person" représente une chaîne littérale.

–"{[a-z]+}" (par ex : toute chaîne littérale fermée par des accolades en plus de {n} et {s}) est interpreté comme une expressoin régulière.

Exemple 1

```
$a = array(
    array('Article' => array('id' => 1, 'title' => 'Article 1')),
    array('Article' => array('id' => 2, 'title' => 'Article 2')),
```

```
        array('Article' => array('id' => 3, 'title' => 'Article 3'))
    );
    $result = Set::classicExtract($a, '{n}.Article.id');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )
    */
    $result = Set::classicExtract($a, '{n}.Article.title');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Article 1
            [1] => Article 2
            [2] => Article 3
        )
    */
    $result = Set::classicExtract($a, '1.Article.title');
    // $result == "Article 2"

    $result = Set::classicExtract($a, '3.Article.title');
    // $result == null
```

Exemple 2

```
    $a = array(
        0 => array('pages' => array('name' => 'page')),
        1 => array('fruites' => array('name' => 'fruit')),
        'test' => array(array('name' => 'jippi')),
        'dot.test' => array(array('name' => 'jippi'))
    );

    $result = Set::classicExtract($a, '{n}.{s}.name');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [0] => page
                )
            [1] => Array
                (
                    [0] => fruit
                )
        )
    */
    $result = Set::classicExtract($a, '{s}.{n}.name');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
```

```
                (
                    [0] => jippi
                )
            [1] => Array
                (
                    [0] => jippi
                )
        )
    */
    $result = Set::classicExtract($a,'{\w+}.{\w+}.name');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [pages] => page
                )
            [1] => Array
                (
                    [fruites] => fruit
                )
            [test] => Array
                (
                    [0] => jippi
                )
            [dot.test] => Array
                (
                    [0] => jippi
                )
        )
    */
    $result = Set::classicExtract($a,'{\d+}.{\w+}.name');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [pages] => page
                )
            [1] => Array
                (
                    [fruites] => fruit
                )
        )
    */
    $result = Set::classicExtract($a,'{n}.{\w+}.name');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [pages] => page
                )
```

```
            [1] => Array
                (
                    [fruites] => fruit
                )
        )
    */
    $result = Set::classicExtract($a,'{s}.{\d+}.name');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [0] => jippi
                )
            [1] => Array
                (
                    [0] => jippi
                )
        )
    */
    $result = Set::classicExtract($a,'{s}');
    /* $result ressemble maintenant à:
        Array
        (

            [0] => Array
                (
                    [0] => Array
                        (
                            [name] => jippi
                        )
                )
            [1] => Array
                (
                    [0] => Array
                        (
                            [name] => jippi
                        )
                )
        )
    */
    $result = Set::classicExtract($a,'{[a-z]}');
    /* $result ressemble maintenant à:
        Array
        (
            [test] => Array
                (
                    [0] => Array
                        (
                            [name] => jippi
                        )
                )
```

```
            [dot.test] => Array
                (
                    [0] => Array
                        (
                            [name] => jippi
                        )
                )
        )
    */
    $result = Set::classicExtract($a, '{dot\.test}.{n}');
    /* $result ressemble maintenant à:
        Array
        (
            [dot.test] => Array
                (
                    [0] => Array
                        (
                            [name] => jippi
                        )
                )
        )
    */
```

static `Set::combine`(*$data*, *$path1 = null*, *$path2 = null*, *$groupPath = null*)

Type retourné array

Crée un tableau associatif utilisant un $path1 comme chemin à build en clé, et en option $path2 comme chemin pour obtenir les valeurs. Si $path2 n'est pas spécifié, toutes les valeurs seront initialisées à null (utile pour Set : :merge). Vous pouvez en option grouper les valeurs par ce qui est obtenu quand on suit le chemin spécifié dans $groupPath. :

```
    $result = Set::combine(array(), '{n}.User.id', '{n}.User.Data');
    // $result == array();

    $result = Set::combine('', '{n}.User.id', '{n}.User.Data');
    // $result == array();

    $a = array(
        array(
            'User' => array(
                'id' => 2,
                'group_id' => 1,
                'Data' => array(
                    'user' => 'mariano.iglesias',
                    'name' => 'Mariano Iglesias'
                )
            )
        ),
        array(
            'User' => array(
                'id' => 14,
                'group_id' => 2,
                'Data' => array(
```

```
                    'user' => 'phpnut',
                    'name' => 'Larry E. Masters'
                )
            )
        ),
        array(
            'User' => array(
                'id' => 25,
                'group_id' => 1,
                'Data' => array(
                    'user' => 'gwoo',
                    'name' => 'The Gwoo'
                )
            )
        )
    );
    $result = Set::combine($a, '{n}.User.id');
    /* $result ressemble maintenant à:
        Array
        (
            [2] =>
            [14] =>
            [25] =>
        )
    */

    $result = Set::combine($a, '{n}.User.id', '{n}.User.non-existant');
    /* $result ressemble maintenant à:
        Array
        (
            [2] =>
            [14] =>
            [25] =>
        )
    */

    $result = Set::combine($a, '{n}.User.id', '{n}.User.Data');
    /* $result ressemble maintenant à:
        Array
        (
            [2] => Array
                (
                    [user] => mariano.iglesias
                    [name] => Mariano Iglesias
                )
            [14] => Array
                (
                    [user] => phpnut
                    [name] => Larry E. Masters
                )
            [25] => Array
                (
                    [user] => gwoo
```

```
                    [name] => The Gwoo
                )
        )
    */

    $result = Set::combine($a, '{n}.User.id', '{n}.User.Data.name');
    /* $result ressemble maintenant à:
        Array
        (
            [2] => Mariano Iglesias
            [14] => Larry E. Masters
            [25] => The Gwoo
        )
    */

    $result = Set::combine($a, '{n}.User.id', '{n}.User.Data', '{n}.User.group_id');
    /* $result ressemble maintenant à:
        Array
        (
            [1] => Array
                (
                    [2] => Array
                        (
                            [user] => mariano.iglesias
                            [name] => Mariano Iglesias
                        )
                    [25] => Array
                        (
                            [user] => gwoo
                            [name] => The Gwoo
                        )
                )
            [2] => Array
                (
                    [14] => Array
                        (
                            [user] => phpnut
                            [name] => Larry E. Masters
                        )
                )
        )
    */

    $result = Set::combine($a, '{n}.User.id', '{n}.User.Data.name', '{n}.User.group_id');
    /* $result ressemble maintenant à:
        Array
        (
            [1] => Array
                (
                    [2] => Mariano Iglesias
                    [25] => The Gwoo
                )
            [2] => Array
```

```
                (
                    [14] => Larry E. Masters
                )
        )
    */

    $result = Set::combine($a, '{n}.User.id', array('{0}: {1}', '{n}.User.Data.user', '{n}
    /* $result ressemble maintenant à:
        Array
        (
            [1] => Array
                (
                    [2] => mariano.iglesias: Mariano Iglesias
                    [25] => gwoo: The Gwoo
                )
            [2] => Array
                (
                    [14] => phpnut: Larry E. Masters
                )
        )
    */

    $result = Set::combine($a, array('{0}: {1}', '{n}.User.Data.user', '{n}.User.Data.name
    /* $result ressemble maintenant à:
        Array
        (
            [mariano.iglesias: Mariano Iglesias] => 2
            [phpnut: Larry E. Masters] => 14
            [gwoo: The Gwoo] => 25
        )
    */

    $result = Set::combine($a, array('{1}: {0}', '{n}.User.Data.user', '{n}.User.Data.name
    /* $result ressemble maintenant à:
        Array
        (
            [Mariano Iglesias: mariano.iglesias] => 2
            [Larry E. Masters: phpnut] => 14
            [The Gwoo: gwoo] => 25
        )
    */

    $result = Set::combine($a, array('%1$s: %2$d', '{n}.User.Data.user', '{n}.User.id'), '

    /* $result ressemble maintenant à:
        Array
        (
            [mariano.iglesias: 2] => Mariano Iglesias
            [phpnut: 14] => Larry E. Masters
            [gwoo: 25] => The Gwoo
        )
    */
```

```
$result = Set::combine($a, array('%2$d: %1$s', '{n}.User.Data.user', '{n}.User.id'), '
/* $result ressemble maintenant à:
    Array
    (
        [2: mariano.iglesias] => Mariano Iglesias
        [14: phpnut] => Larry E. Masters
        [25: gwoo] => The Gwoo
    )
*/
```

static `Set::contains` (*$val1*, *$val2 = null*)

Type retourné boolean

Detérmine si un Set ou un tableau contient les clés exactes et les valeurs d'un autre :

```
$a = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about')
);
$b = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about'),
    2 => array('name' => 'contact'),
    'a' => 'b'
);

$result = Set::contains($a, $a);
// True
$result = Set::contains($a, $b);
// False
$result = Set::contains($b, $a);
// True
```

static `Set::countDim` (*$array = null*, *$all = false*, *$count = 0*)

Type retourné integer

Compte les dimensions d'un tableau. Si $all est défini à false (qui est la valeur par défaut) il va seulement considérer la dimension du premier élément dans le tableau :

```
$data = array('one', '2', 'three');
$result = Set::countDim($data);
// $result == 1

$data = array('1' => '1.1', '2', '3');
$result = Set::countDim($data);
// $result == 1

$data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => '3.1.1'));
$result = Set::countDim($data);
// $result == 2

$data = array('1' => '1.1', '2', '3' => array('3.1' => '3.1.1'));
$result = Set::countDim($data);
// $result == 1
```

```
$data = array('1' => '1.1', '2', '3' => array('3.1' => '3.1.1'));
$result = Set::countDim($data, true);
// $result == 2

$data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => array('3.1.1'
$result = Set::countDim($data);
// $result == 2

$data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' => array('3.1.1'
$result = Set::countDim($data, true);
// $result == 3

$data = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' => array('2.1.1
$result = Set::countDim($data, true);
// $result == 4

$data = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' => array('2.1.1
$result = Set::countDim($data, true);
// $result == 5

$data = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' => array('2.1.1
$result = Set::countDim($data, true);
// $result == 5

$set = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' => array('2.1.1'
$result = Set::countDim($set, false, 0);
// $result == 2

$result = Set::countDim($set, true);
// $result == 5
```

static `Set::diff` (*$val1*, *$val2 = null*)

Type retourné array

Compute la différence entre un Set et un tableau, deux Sets, ou deux tableaux :

```
$a = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about')
);
$b = array(
    0 => array('name' => 'main'),
    1 => array('name' => 'about'),
    2 => array('name' => 'contact')
);

$result = Set::diff($a, $b);
/* $result ressemble maintenant à:
    Array
    (
        [2] => Array
            (
                [name] => contact
```

```
                )
        )
    */
    $result = Set::diff($a, array());
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [name] => main
                )
            [1] => Array
                (
                    [name] => about
                )
        )
    */
    $result = Set::diff(array(), $b);
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [name] => main
                )
            [1] => Array
                (
                    [name] => about
                )
            [2] => Array
                (
                    [name] => contact
                )
        )
    */

    $b = array(
        0 => array('name' => 'me'),
        1 => array('name' => 'about')
    );

    $result = Set::diff($a, $b);
    /* $result now looks like:
        Array
        (
            [0] => Array
                (
                    [name] => main
                )
        )
    */
```

static `Set::enum`(*$select*, *$list=null*)

Type retourné string

La méthode enum fonctionne bien quand on utilise les éléments html select. Elle retourne une valeur d'un tableau listé si la clé existe.

Si un $list séparé par des virgules est passé dans les tableaux sont numériques avec la clé allant de 0 $list = 'no, yes' traduirait à $list = array(0 => 'no', 1 => 'yes') ;

Si un tableau est utilisé, les clés peuvent être des chaînes exemple : array('no' => 0,'yes' => 1) ;

$list par défaut à 0 = no 1 = yes si param n'est pas passé :

```
$res = Set::enum(1, 'one, two');
// $res est 'two'

$res = Set::enum('no', array('no' => 0, 'yes' => 1));
// $res est 0

$res = Set::enum('first', array('first' => 'one', 'second' => 'two'));
// $res est 'one'
```

static `Set::extract` (*$path*, *$data=null*, *$options=array()*)

Type retourné array

Set : :extract utilise la syntaxe basique XPath 2.0 pour retourner les sous-ensembles de vos données à partir d'un fin ou d'un find all. Cette fonction vous permet de récupèrer vos données rapidement sans avoir à boucler à travers des tableaux multi-dimensionnels ou de traverser à travers les structures en arbre.

Note : Si `$path` ne contient pas un '/', l'appel sera délégué à `Set::classicExtract()`

```
// Utilisation habituelle:
$users = $this->User->find("all");
$results = Set::extract('/User/id', $users);
// results retourne:
// array(1,2,3,4,5,...);
```

Les sélecteurs implémentés actuellement :

Selector	Note
/User/id	Similaire au {n}.User.id classique
/User[2]/name	Sélectionne le nom du deuxième User
/User[id<2]	Sélectionne tous les Users avec un id < 2
/User[id>2][<5]	Sélectionne tous les Users avec un id > 2 mais 5
/Post/Comment[author=john]/.name	Sélectionne le nom de tous les Posts qui ont au moins un Comment écrit par john
/Posts[title]	Sélectionne tous les Posts qui ont une clé 'title'
/Comment/.[1]	Sélectionne les contenus du premier contenu
/Comment/.[:last]	Sélectionne le dernier comment
/Comment/.[:first]	Sélectionne le premier comment
/Com-ment[text=/cakephp/i]	Sélectionne tous les comments qui ont un texte correspondant au regex /cakephp/i
/Comment/@*	Sélectionne les noms de clé de tous les comments. Actuellement seuls les chemins absolus commançant par un unique '/' sont supportés. Merci de reporter tout bug si vous en trouvez. Les suggestions pour des fonctionnalités supplémentaires sont bienvenues additional features are welcome.

Pour en apprendre plus sur Set : :extract() référez vous à la fonction testExtract() dans `/lib/Cake/Test/Case/Utility/SetTest.php`.

static `Set::filter`(*$var*, *$isArray=null*)

Type retourné array

Filtre les éléments vide d'un tableau route, en excluant '0' :

```
$res = Set::filter(array('0', false, true, 0, array('one thing', 'I can tell you', 'is

/* $res ressemble maintenant à:
    Array (
        [0] => 0
        [2] => 1
        [3] => 0
        [4] => Array
            (
                [0] => one thing
                [1] => I can tell you
                [2] => is you got to be
            )
    )
*/
```

static `Set::flatten`(*$data*, *$separator='.'*)

Type retourné array

Transforme un tableau multi-dimensional en un tableau à dimension unique :

```
$arr = array(
    array(
        'Post' => array('id' => '1', 'title' => 'First Post'),
        'Author' => array('id' => '1', 'user' => 'Kyle'),
    ),
```

```
        array(
            'Post' => array('id' => '2', 'title' => 'Second Post'),
            'Author' => array('id' => '3', 'user' => 'Crystal'),
        ),
    );
    $res = Set::flatten($arr);
    /* $res ressemble maintenant à:
        Array (
            [0.Post.id] => 1
            [0.Post.title] => First Post
            [0.Author.id] => 1
            [0.Author.user] => Kyle
            [1.Post.id] => 2
            [1.Post.title] => Second Post
            [1.Author.id] => 3
            [1.Author.user] => Crystal
        )
    */
```

static `Set::format` (*$data*, *$format*, *$keys*)

Type retourné array

Retourne une série de valeurs extraites d'un tableau, formaté en un format de chaîne :

```
    $data = array(
        array('Person' => array('first_name' => 'Nate', 'last_name' => 'Abele', 'city' =>
        array('Person' => array('first_name' => 'Larry', 'last_name' => 'Masters', 'city'
        array('Person' => array('first_name' => 'Garrett', 'last_name' => 'Woodworth', 'ci

    $res = Set::format($data, '{1}, {0}', array('{n}.Person.first_name', '{n}.Person.last_
    /*
    Array
    (
        [0] => Abele, Nate
        [1] => Masters, Larry
        [2] => Woodworth, Garrett
    )
    */

    $res = Set::format($data, '{0}, {1}', array('{n}.Person.city', '{n}.Person.state'));
    /*
    Array
    (
        [0] => Boston, MA
        [1] => Boondock, TN
        [2] => Venice Beach, CA
    )
    */
    $res = Set::format($data, '{{0}, {1}}', array('{n}.Person.city', '{n}.Person.state'));
    /*
    Array
    (
        [0] => {Boston, MA}
        [1] => {Boondock, TN}
```

```
        [2] => {Venice Beach, CA}
    )
    */
    $res = Set::format($data, '{%2$d, %1$s}', array('{n}.Person.something', '{n}.Person.so
    /*
    Array
    (
        [0] => {42, 42}
        [1] => {0, {0}}
        [2] => {0, {1}}
    )
    */
    $res = Set::format($data, '%2$d, %1$s', array('{n}.Person.first_name', '{n}.Person.som
    /*
    Array
    (
        [0] => 42, Nate
        [1] => 0, Larry
        [2] => 0, Garrett
    )
    */
    $res = Set::format($data, '%1$s, %2$d', array('{n}.Person.first_name', '{n}.Person.som
    /*
    Array
    (
        [0] => Nate, 42
        [1] => Larry, 0
        [2] => Garrett, 0
    )
    */
```

static `Set::insert` (*$list*, *$path*, *$data = null*)

Type retourné array

Insére $data dans un tableau comme défini dans $path. :

```
    $a = array(
        'pages' => array('name' => 'page')
    );
    $result = Set::insert($a, 'files', array('name' => 'files'));
    /* $result ressemble maintenant à:
        Array
        (
            [pages] => Array
                (
                    [name] => page
                )
            [files] => Array
                (
                    [name] => files
                )
        )
    */
```

```
$a = array(
    'pages' => array('name' => 'page')
);
$result = Set::insert($a, 'pages.name', array());
/* $result ressemble maintenant à:
    Array
    (
        [pages] => Array
            (
                [name] => Array
                    (
                    )
            )
    )
*/

$a = array(
    'pages' => array(
        0 => array('name' => 'main'),
        1 => array('name' => 'about')
    )
);
$result = Set::insert($a, 'pages.1.vars', array('title' => 'page title'));
/* $result ressemble maintenant à:
    Array
    (
        [pages] => Array
            (
                [0] => Array
                    (
                        [name] => main
                    )
                [1] => Array
                    (
                        [name] => about
                        [vars] => Array
                            (
                                [title] => page title
                            )
                    )
            )
    )
*/
```

static `Set::map` (*$class = 'stdClass'*, *$tmp = 'stdClass'*)

Type retourné object

Cette méthode Mappe le contenu de l'objet Set en un objet hiérarchisé et maintient les clés numériques en tableaux d'objets.

Basiquement, la fonction map transforme le tableau d'items en classe d'objets initialisée. Par défaut il transforme un tableau en un Objet stdClass, cependant vous pouvez mapper les valeurs en un type de classe. Exemple : Set : :map($array_of_values, 'nameOfYourClass') ; :

```
$data = array(
    array(
        "IndexedPage" => array(
            "id" => 1,
            "url" => 'http://blah.com/',
            'hash' => '68a9f053b19526d08e36c6a9ad150737933816a5',
            'get_vars' => '',
            'redirect' => '',
            'created' => "1195055503",
            'updated' => "1195055503",
        )
    ),
    array(
        "IndexedPage" => array(
            "id" => 2,
            "url" => 'http://blah.com/',
            'hash' => '68a9f053b19526d08e36c6a9ad150737933816a5',
            'get_vars' => '',
            'redirect' => '',
            'created' => "1195055503",
            'updated' => "1195055503",
        ),
    )
);
$mapped = Set::map($data);

/* $mapped ressemble maintenant à:

    Array
    (
        [0] => stdClass Object
            (
                [_name_] => IndexedPage
                [id] => 1
                [url] => http://blah.com/
                [hash] => 68a9f053b19526d08e36c6a9ad150737933816a5
                [get_vars] =>
                [redirect] =>
                [created] => 1195055503
                [updated] => 1195055503
            )

        [1] => stdClass Object
            (
                [_name_] => IndexedPage
                [id] => 2
                [url] => http://blah.com/
                [hash] => 68a9f053b19526d08e36c6a9ad150737933816a5
                [get_vars] =>
                [redirect] =>
                [created] => 1195055503
                [updated] => 1195055503
            )
```

```
        )

    */
```

Utilisation de Set : :map() avec une classe personnalisée en second paramètre :

```
    class MyClass {
        public function sayHi() {
            echo 'Hi!';
        }
    }

    $mapped = Set::map($data, 'MyClass');
    //Maintenant vous pouvez accéder à toutes les propriétés comme dans
    //l'exemple ci-dessus, mais aussi vous pouvez appeler les méthodes
    //MyClass
    $mapped->[0]->sayHi();
```

static `Set::matches` (*$conditions*, *$data=array()*, *$i = null*, *$length=null*)

Type retourné boolean

Set : :matches peut être utilisé pour voir si un item unique ou un xpath donné admet certaines conditions. :

```
    $a = array(
        array('Article' => array('id' => 1, 'title' => 'Article 1')),
        array('Article' => array('id' => 2, 'title' => 'Article 2')),
        array('Article' => array('id' => 3, 'title' => 'Article 3')));
    $res=Set::matches(array('id>2'), $a[1]['Article']);
    // retourne false
    $res=Set::matches(array('id>=2'), $a[1]['Article']);
    // retourne true
    $res=Set::matches(array('id>=3'), $a[1]['Article']);
    // retourne false
    $res=Set::matches(array('id<=2'), $a[1]['Article']);
    // retourne true
    $res=Set::matches(array('id<2'), $a[1]['Article']);
    // retourne false
    $res=Set::matches(array('id>1'), $a[1]['Article']);
    // retourne true
    $res=Set::matches(array('id>1', 'id<3', 'id!=0'), $a[1]['Article']);
    // retourne true
    $res=Set::matches(array('3'), null, 3);
    // retourne true
    $res=Set::matches(array('5'), null, 5);
    // retourne true
    $res=Set::matches(array('id'), $a[1]['Article']);
    // retourne true
    $res=Set::matches(array('id', 'title'), $a[1]['Article']);
    // retourne true
    $res=Set::matches(array('non-existent'), $a[1]['Article']);
    // retourne false
    $res=Set::matches('/Article[id=2]', $a);
```

```
// retourne true
$res=Set::matches('/Article[id=4]', $a);
// retourne false
$res=Set::matches(array(), $a);
// retourne true
```

static `Set::merge`(*$arr1*, *$arr2=null*)

Type retourné array

Cette fonction peut être imaginée comme un hybride entre array_merge et array_merge_recursive de PHP. La différence entre les deux est que si une clé de tableau contient un autre tableau alors la fonction se comporte de façon récursive (pas comme array_merge) mais le ne fait pas pour les clés contenant des chaînes (pas comme array_merge_recursive). Regardez le test unitaire pour plus d'informations.

Note : Cette fonction va fonctionner avec un montant illimité d'arguments et de paramètres non-tableaux typecasts dans des tableaux.

```
$arry1 = array(
    array(
        'id' => '48c2570e-dfa8-4c32-a35e-0d71cbdd56cb',
        'name' => 'mysql raleigh-workshop-08 < 2008-09-05.sql ',
        'description' => 'Importing an sql dump'
    ),
    array(
        'id' => '48c257a8-cf7c-4af2-ac2f-114ecbdd56cb',
        'name' => 'pbpaste | grep -i Unpaid | pbcopy',
        'description' => 'Remove all lines that say "Unpaid".',
    )
);
$arry2 = 4;
$arry3 = array(0 => "test array", "cats" => "dogs", "people" => 1267);
$arry4 = array("cats" => "felines", "dog" => "angry");
$res = Set::merge($arry1, $arry2, $arry3, $arry4);

/* $res ressemble maintenant à:
Array
(
    [0] => Array
        (
            [id] => 48c2570e-dfa8-4c32-a35e-0d71cbdd56cb
            [name] => mysql raleigh-workshop-08 < 2008-09-05.sql
            [description] => Importing an sql dump
        )

    [1] => Array
        (
            [id] => 48c257a8-cf7c-4af2-ac2f-114ecbdd56cb
            [name] => pbpaste | grep -i Unpaid | pbcopy
            [description] => Retire toutes les lignes qui disent "Unpaid".
        )
```

```
        [2] => 4
        [3] => test array
        [cats] => felines
        [people] => 1267
        [dog] => angry
    )
    */
```

static `Set::normalize`(*$list*, *$assoc* = *true*, *$sep* = ', ', *$trim* = *true*)

Type retourné array

Normalise une liste de chaîne ou de tableau. :

```
    $a = array('Tree', 'CounterCache',
            'Upload' => array(
                'folder' => 'products',
                'fields' => array('image_1_id', 'image_2_id', 'image_3_id', 'image_4_id',
    $b =  array('Cacheable' => array('enabled' => false),
            'Limit',
            'Bindable',
            'Validator',
            'Transactional');
    $result = Set::normalize($a);
    /* $result ressemble maintenant à:
        Array
        (
            [Tree] =>
            [CounterCache] =>
            [Upload] => Array
                (
                    [folder] => products
                    [fields] => Array
                        (
                            [0] => image_1_id
                            [1] => image_2_id
                            [2] => image_3_id
                            [3] => image_4_id
                            [4] => image_5_id
                        )
                )
        )
    */
    $result = Set::normalize($b);
    /* $result ressemble maintenant à:
        Array
        (
            [Cacheable] => Array
                (
                    [enabled] =>
                )

            [Limit] =>
            [Bindable] =>
            [Validator] =>
```

```
            [Transactional] =>
        )
    */
    $result = Set::merge($a, $b); // Fusionne maintenant les deux et normalize
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Tree
            [1] => CounterCache
            [Upload] => Array
                (
                    [folder] => products
                    [fields] => Array
                        (
                            [0] => image_1_id
                            [1] => image_2_id
                            [2] => image_3_id
                            [3] => image_4_id
                            [4] => image_5_id
                        )

                )
            [Cacheable] => Array
                (
                    [enabled] =>
                )
            [2] => Limit
            [3] => Bindable
            [4] => Validator
            [5] => Transactional
        )
    */
    $result = Set::normalize(Set::merge($a, $b));
    /* $result ressemble maintenant à:
        Array
        (
            [Tree] =>
            [CounterCache] =>
            [Upload] => Array
                (
                    [folder] => products
                    [fields] => Array
                        (
                            [0] => image_1_id
                            [1] => image_2_id
                            [2] => image_3_id
                            [3] => image_4_id
                            [4] => image_5_id
                        )

                )
            [Cacheable] => Array
                (
```

```
                    [enabled] =>
                )
            [Limit] =>
            [Bindable] =>
            [Validator] =>
            [Transactional] =>
        )
    */
```

static `Set::numeric`(*$array=null*)

Type retourné boolean

Vérifie si toutes les valeurs dans le tableau sont numériques :

```
$data = array('one');
$res = Set::numeric(array_keys($data));

// $res est true

$data = array(1 => 'one');
$res = Set::numeric($data);

// $res est false

$data = array('one');
$res = Set::numeric($data);

// $res est false

$data = array('one' => 'two');
$res = Set::numeric($data);

// $res est false

$data = array('one' => 1);
$res = Set::numeric($data);

// $res est true

$data = array(0);
$res = Set::numeric($data);

// $res est true

$data = array('one', 'two', 'three', 'four', 'five');
$res = Set::numeric(array_keys($data));

// $res est true

$data = array(1 => 'one', 2 => 'two', 3 => 'three', 4 => 'four', 5 => 'five');
$res = Set::numeric(array_keys($data));

// $res est true
```

```
$data = array('1' => 'one', 2 => 'two', 3 => 'three', 4 => 'four', 5 => 'five');
$res = Set::numeric(array_keys($data));

// $res est true

$data = array('one', 2 => 'two', 3 => 'three', 4 => 'four', 'a' => 'five');
$res = Set::numeric(array_keys($data));

// $res est false
```

static `Set::pushDiff` (*$array1*, *$array2*)

Type retourné array

Cette fonction fusionne deux tableaux et pousse les différences dans array2 à la fin du tableau résultant.

Exemple 1

```
$array1 = array('ModelOne' => array('id' => 1001, 'field_one' => 'a1.m1.f1', 'field_tw
$array2 = array('ModelOne' => array('id' => 1003, 'field_one' => 'a3.m1.f1', 'field_tw
$res = Set::pushDiff($array1, $array2);

/* $res ressemble maintenant à:
    Array
    (
        [ModelOne] => Array
            (
                [id] => 1001
                [field_one] => a1.m1.f1
                [field_two] => a1.m1.f2
                [field_three] => a3.m1.f3
            )
    )
*/
```

Exemple 2

```
$array1 = array("a" => "b", 1 => 20938, "c" => "string");
$array2 = array("b" => "b", 3 => 238, "c" => "string", array("extra_field"));
$res = Set::pushDiff($array1, $array2);
/* $res ressemble maintenant à:
    Array
    (
        [a] => b
        [1] => 20938
        [c] => string
        [b] => b
        [3] => 238
        [4] => Array
            (
                [0] => extra_field
            )
    )
*/
```

static `Set::remove`(*$list*, *$path = null*)

Type retourné array

Retire un élément d'un Set ou d'un tableau selon ce qui est défini par $path :

```
$a = array(
    'pages'      => array('name' => 'page'),
    'files'      => array('name' => 'files')
);

$result = Set::remove($a, 'files');
/* $result ressemble maintenant à:
    Array
    (
        [pages] => Array
            (
                [name] => page
            )

    )
*/
```

static `Set::reverse`(*$object*)

Type retourné array

Set : :reverse est au fond l'opposé de `Set::map`. Elle convertit un objet en un tableau. Si $object n'est pas un objet, reverse va simplement retourner $object. :

```
$result = Set::reverse(null);
// Null
$result = Set::reverse(false);
// false
$a = array(
    'Post' => array('id' => 1, 'title' => 'Premier Post'),
    'Comment' => array(
        array('id' => 1, 'title' => 'Premier Comment'),
        array('id' => 2, 'title' => 'Deuxième Comment')
    ),
    'Tag' => array(
        array('id' => 1, 'title' => 'Premier Tag'),
        array('id' => 2, 'title' => 'Deuxième Tag')
    ),
);
$map = Set::map($a); // Change $a dans une classe object
/* $map ressemble maintenant à:
    stdClass Object
    (
        [_name_] => Post
        [id] => 1
        [title] => Premier Post
        [Comment] => Array
            (
                [0] => stdClass Object
                    (
```

```
                            [id] => 1
                            [title] => Premier Comment
                        )
                    [1] => stdClass Object
                        (
                            [id] => 2
                            [title] => Deuxième Comment
                        )
                )
            [Tag] => Array
                (
                    [0] => stdClass Object
                        (
                            [id] => 1
                            [title] => Premier Tag
                        )
                    [1] => stdClass Object
                        (
                            [id] => 2
                            [title] => Deuxième Tag
                        )
                )
        )
    */

    $result = Set::reverse($map);
    /* $result ressemble maintenant à:
        Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => Premier Post
                    [Comment] => Array
                        (
                            [0] => Array
                                (
                                    [id] => 1
                                    [title] => Premier Comment
                                )
                            [1] => Array
                                (
                                    [id] => 2
                                    [title] => Deuxième Comment
                                )
                        )
                    [Tag] => Array
                        (
                            [0] => Array
                                (
                                    [id] => 1
                                    [title] => First Tag
                                )
```

```
                            [1] => Array
                                (
                                    [id] => 2
                                    [title] => Second Tag
                                )
                        )
                )
        )
    */

    $result = Set::reverse($a['Post']); // Retourne juste un tableau
    /* $result ressemble maintenant à:
        Array
        (
            [id] => 1
            [title] => Premier Post
        )
    */
```

static `Set::sort`(*$data*, *$path*, *$dir*)

Type retourné array

Trie un tableau selon toute valeur, determiné par un chemin Set-compatible :

```
    $a = array(
        0 => array('Person' => array('name' => 'Jeff')),
        1 => array('Shirt' => array('color' => 'black'))
    );
    $result = Set::sort($a, '{n}.Person.name', 'asc');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [Shirt] => Array
                        (
                            [color] => black
                        )
                )
            [1] => Array
                (
                    [Person] => Array
                        (
                            [name] => Jeff
                        )
                )
        )
    */

    $result = Set::sort($a, '{n}.Shirt', 'asc');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
```

```
                (
                    [Person] => Array
                        (
                            [name] => Jeff
                        )
                )
            [1] => Array
                (
                    [Shirt] => Array
                        (
                            [color] => black
                        )
                )
        )
    */

    $result = Set::sort($a, '{n}', 'desc');
    /* $result ressemble maintenant à:
        Array
        (
            [0] => Array
                (
                    [Shirt] => Array
                        (
                            [color] => black
                        )
                )
            [1] => Array
                (
                    [Person] => Array
                        (
                            [name] => Jeff
                        )
                )
        )
    */

    $a = array(
        array(7,6,4),
        array(3,4,5),
        array(3,2,1),
    );
```

static `Set::apply` (*$path*, *$array*, *$callback*, *$options = array()*)

Type retourné mixed

Applique un callback aux éléments d'un tableau extait par un chemin compatible Set : :extract :

```
                    $data = array(
        array('Movie' => array('id' => 1, 'title' => 'movie 3', 'rating' => 5)),
        array('Movie' => array('id' => 1, 'title' => 'movie 1', 'rating' => 1)),
        array('Movie' => array('id' => 1, 'title' => 'movie 2', 'rating' => 3)),
    );
```

```
$result = Set::apply('/Movie/rating', $data, 'array_sum');
// résultat égal à 9

$result = Set::apply('/Movie/title', $data, 'strtoupper', array('type' => 'map'));
// résultat égal à array('MOVIE 3', 'MOVIE 1', 'MOVIE 2')
// $options sont: - type : peut être 'pass' utilise call_user_func_array(), 'map' util
```

static `Set::nest` (*$data*, *$options = array()*)

Type retourné array

Prend un tableau plat et retourne un tableau imbriqué :

```
$data = array(
    array('ModelName' => array('id' => 1, 'parent_id' => null)),
    array('ModelName' => array('id' => 2, 'parent_id' => 1)),
    array('ModelName' => array('id' => 3, 'parent_id' => 1)),
    array('ModelName' => array('id' => 4, 'parent_id' => 1)),
    array('ModelName' => array('id' => 5, 'parent_id' => 1)),
    array('ModelName' => array('id' => 6, 'parent_id' => null)),
    array('ModelName' => array('id' => 7, 'parent_id' => 6)),
    array('ModelName' => array('id' => 8, 'parent_id' => 6)),
    array('ModelName' => array('id' => 9, 'parent_id' => 6)),
    array('ModelName' => array('id' => 10, 'parent_id' => 6))
);

$result = Set::nest($data, array('root' => 6));
/* $result ressemble maintenant à:
    array(
            (int) 0 => array(
                    'ModelName' => array(
                            'id' => (int) 6,
                            'parent_id' => null
                    ),
                    'children' => array(
                            (int) 0 => array(
                                    'ModelName' => array(
                                            'id' => (int) 7,
                                            'parent_id' => (int) 6
                                    ),
                                    'children' => array()
                            ),
                            (int) 1 => array(
                                    'ModelName' => array(
                                            'id' => (int) 8,
                                            'parent_id' => (int) 6
                                    ),
                                    'children' => array()
                            ),
                            (int) 2 => array(
                                    'ModelName' => array(
                                            'id' => (int) 9,
                                            'parent_id' => (int) 6
                                    ),
                                    'children' => array()
```

```
                                    ),
                                    (int) 3 => array(
                                            'ModelName' => array(
                                                    'id' => (int) 10,
                                                    'parent_id' => (int) 6
                                            ),
                                            'children' => array()
                                    )
                            )
                    )
            )
    */
```

String

class `String`

La classe String inclut des méthodes pratiques pour la création et la manipulation des chaînes de caractères et est normalement accessible statiquement. Exemple : `String::uuid()`.

Si vous avez besoin des fonctionnalités de `TextHelper` en-dehors d'une `View`, utilisez la classe `String` :

```
class UsersController extends AppController {

    public $components = array('Auth');

    public function afterLogin() {
        App::uses('String', 'Utility');
        $message = $this->User->find('new_message');
        if (!empty($message)) {
            // notifie à l'utilisateur d'un nouveau message
            $this->Session->setFlash(__('Vous avez un message: %s', String::truncate($messa
        }
    }
}
```

Modifié dans la version 2.1 : Plusieurs méthodes de `TextHelper` ont été déplacées dans la classe `String`.

static `String::uuid`

La méthode uuid est utilisée pour générer des identificateurs uniques comme per **RFC 4122** (http ://tools.ietf.org/html/rfc4122.html). uuid est une chaîne de caractères de 128bit au format 485fc381-e790-47a3-9794-1337c0a8fe68.

```
String::uuid(); // 485fc381-e790-47a3-9794-1337c0a8fe68
```

static `String::tokenize`(*$data*, *$separator* = ', ', *$leftBound* = '(', *$rightBound* = ')')

Tokenizes une chaîne en utilisant `$separator`, en ignorant toute instance de `$separator` qui apparait entre `$leftBound` et `$rightBound`.

Cette méthode peut être utile quand on sépare les données en formatage régulier comme les listes de tag :

```
$data = "cakephp 'great framework' php";
$result = String::tokenize($data, ' ', "'", "'");
// le résultat contient
array('cakephp', "'great framework'", 'php');
```

static `String::insert` (*$string*, *$data*, *$options = array()*)

La méthode insérée est utilisée pour créer des chaînes templates et pour permettre les remplacements de clé/valeur :

```
String::insert('Mon nom est :name et j'ai :age ans.', array('name' => 'Bob', 'age' =>
// génére: "Mon nom est Bob et j'ai 65 ans."
```

static `String::cleanInsert` (*$string*, *$options = array()*)

Nettoie une chaîne formatée `String::insert` avec $options donnée qui dépend de la clé 'clean' dans $options. La méthode par défaut utilisée est le texte mais html est aussi disponible. Le but de cette fonction est de remplacer tous les espaces blancs et les balises non nécessaires autour des placeholders qui ne sont pas remplacés par Set : :insert.

Vous pouvez utiliser les options suivantes dans le tableau options :

```
$options = array(
    'clean' => array(
        'method' => 'text', // ou html
    ),

    'before' => '',
    'after' => ''
);
```

static `String::wrap` (*$text*, *$options = array()*)

Entoure un block de texte pour un ensemble de largeur, et indente aussi les blocks. Peut entourer intelligemment le texte ainsi les mots ne sont pas sliced across lines :

```
$text = 'Ceci est la chanson qui ne stoppe jamais.';
$result = String::wrap($text, 22);

// retourne
Ceci est la chanson
qui ne stoppe jamais.
```

Vous pouvez fournir un tableau d'options qui contrôlent la façon dont on entoure. Les options possibles sont :

–`width` La largeur de l'enroulement. Par défaut à 72.

–`wordWrap` Entoure ou non les mots entiers. Par défaut à true.

–`indent` Le caractère avec lequel on indente les lignes. Par défaut à ''.

–`indentAt` Le nombre de ligne pour commencer l'indentation du texte. Par défaut à 0.

`String::highlight` (*string $haystack*, *string $needle*, *array $options = array()*)

Paramètres

- **$haystack** (*string*) – La chaîne de caractères à rechercher.
- **$needle** (*string*) – La chaîne à trouver.
- **$options** (*array*) – Un tableau d'options, voir ci-dessous.

Mettre en avant `$needle` dans `$haystack` en utilisant la chaîne spécifique `$options['format']` ou une chaîne par défaut.

Options :

–'format' - chaîne la partie de html avec laquelle la phrase sera mise en excergue.

–'html' - bool Si true, va ignorer tous les tags HTML, s'assurant que seul le bon texte est mise en avant.

Exemple :

```
// appelé avec TextHelper
echo $this->Text->highlight($last_sentence, 'using', array('format' => '<span class="h

// appelé avec String
App::uses('String', 'Utility');
echo String::highlight($last_sentence, 'using', array('format' => '<span class="highli
```

Sortie :

```
Highlights $needle in $haystack <span class="highlight">using</span>
the $options['format'] string specified  or a default string.
```

`String::stripLinks`(*$text*)

Enlève le `$text` fourni de tout lien HTML.

`String::truncate`(*string $text*, *int $length=100*, *array $options*)

Paramètres

- **$text** (*string*) – Le texte à tronquer.
- **$length** (*int*) – La longueur de trim.
- **$options** (*array*) – Un tableau d'options à utiliser.

Coupe une chaîne avec `$length` et ajoute un suffixe avec `'ending'` si le texte est plus long que `$length`. Si `'exact'` est passé à `false`, le truchement va se faire après le mot de fin suivant. Si `'html'` est passé à `true`, les balises html seront respectés et ne seront pas coupés.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et a les clés suivantes possibles par défaut, celles-ci étant toutes optionnelles :

```
array(
    'ellipsis' => '...',
    'exact' => true,
    'html' => false
)
```

Exemple :

```
// appelé avec TextHelper
echo $this->Text->truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    array(
        'ellipsis' => '...',
        'exact' => false
    )
);
```

```
// appelé avec String
App::uses('String', 'Utility');
echo String::truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    array(
        'ellipsis' => '...',
        'exact' => false
    )
);
```

Sortie :

```
The killer crept...
```

Modifié dans la version 2.3 : `ending` a été remplacé par `ellipsis`. `ending` est toujours utilisé dans 2.2.1.

`String::tail` (*string $text*, *int $length=100*, *array $options*)

Paramètres

- **$text** (*string*) – The text à tronquer.
- **$length** (*int*) – La longueur de trim.
- **$options** (*array*) – Un tableau d'options à utiliser.

Coupe une chaîne à la longueur `$length` et ajoute un préfixe avec `'ellipsis'` si le texte est plus long que `$length`. Si `'exact'` est passé à `false`, le truchement se fera avant le prochain mot de fin.

`$options` est utilisé pour passer tous les paramètres supplémentaires, et a les clés possibles suivantes par défaut, toutes sont optionnelles :

```
array(
    'ellipsis' => '...',
    'exact' => true
)
```

Introduit dans la version 2.3. Exemple :

```
// appelé avec TextHelper
echo $this->Text->tail(
    'I packed my bag and in it I put a PSP, a PS3, a TV, a C# program that can divide
    70,
    array(
        'ellipsis' => '...',
        'exact' => false
    )
);

// appelé avec String
App::uses('String', 'Utility');
echo String::tail(
    'I packed my bag and in it I put a PSP, a PS3, a TV, a C# program that can divide
    70,
    array(
```

```
            'ellipsis' => '...',
            'exact' => false
        )
    );
```

Sortie :

```
...a TV, a C# program that can divide by zero, death metal t-shirts
```

`String::excerpt` (*string $haystack*, *string $needle*, *integer $radius=100*, *string $ending="..."*)

Paramètres

- **$haystack** (*string*) – La chaîne à chercher.
- **$needle** (*string*) – La chaîne to excerpt around.
- **$radius** (*int*) – Le nombre de caractères de chaque côté de $needle que vous souhaitez inclure.
- **$ending** (*string*) – Le Texte à ajouter/préfixer au début ou à la fin du résultat.

Extrait un excerpt de `$haystack` surrounding the `$needle` avec un nombre de caractères de chaque côté determiné par `$radius`, and prefix/suffix with `$ending`. Cette méthode est spécialement pratique pour les résultats recherchés. La chaîne requêtée ou les mots clés peuvent être montrés dans le document résultant. :

```
// appelé avec TextHelper
echo $this->Text->excerpt($last_paragraph, 'method', 50, '...');

// appelé avec String
App::uses('String', 'Utility');
echo String::excerpt($last_paragraph, 'method', 50, '...');
```

Sortie :

```
... par $radius, et prefix/suffix avec $ending. Cette méthode est
spécialement pratique pour les résultats de recherche. La requête...
```

`String::toList` (*array $list*, *$and='and'*)

Paramètres

- **$list** (*array*) – Tableau d'éléments à combiner dans une list sentence.
- **$and** (*string*) – Le mot utilisé pour le dernier join.

Crée une liste séparée avec des virgules, où les deux derniers items sont joins avec 'and'. :

```
// appelé avec TextHelper
echo $this->Text->toList($colors);

// appelé avec String
App::uses('String', 'Utility');
echo String::toList($colors);
```

Sortie :

```
red, orange, yellow, green, blue, indigo et violet
```

Xml

class `Xml`

La classe Xml a été reconstruite. Comme PHP 5 a SimpleXML (http ://php.net/simplexml) et DOMDocument (http ://php.net/domdocument), CakePHP ne nécéssite pas de ré-implémenter un parser XML. La nouvelle classe XML va fondamentalement transformer un tableau en objets SimpleXMLElement ou DOMDocument, et vice versa.

Importer les données vers la classe Xml

Dans CakePHP 1.3, vous pouviez passez les tableaux, les XML et les chaînes de caractère, les URL ou les chemins de fichier vers le constructeur de la classe Xml pour importer les données. Dans CakePHP 2.0, vous pouvez le faire en utilisant `Xml::build()`. A moins que le retour soit un objet Xml, cela retournera un objet SimpleXMLElement ou DOMDocument (selon votre paramètre options - par défault SimpleXMLElement). Ci-dessous les échantillons sur la façon d'importer des données depuis une URL :

```
// Vieille méthode:
$xml = new Xml('http://bakery.cakephp.org/articles/rss');

// Nouvelle méthode en utilisant SimpleXML
$xml = Xml::build('http://bakery.cakephp.org/articles/rss');
// $xml est maintenant une instance de SimpleXMLElement

//ou
$xml = Xml::build('http://bakery.cakephp.org/articles/rss', array('return' => 'simplexml'))
// $xml est maintenant une instance de SimpleXMLElement

// Nouvelle méthode en utilisant DOMDocument
$xml = Xml::build('http://bakery.cakephp.org/articles/rss', array('return' => 'domdocument'
// $xml est maintenant une instance de DOMDocument
```

Vous pouvez utiliser `Xml::build()` pour construire les objets XML à partir de diverses sources. Vous pouvez utiliser XML pour construire des objets à partir d'une chaîne de caractère :

```
$text = '<?xml version="1.0" encoding="utf-8"?>
<post>
    <id>1</id>
    <title>Meilleur post</title>
    <body> ... </body>
</post>';
$xml = Xml::build($text);
```

Vous pouvez aussi construire des objets Xml à partir de fichiers locaux, ou de fichiers distants. Les fichiers distants seront récupérés avec `HttpSocket` :

```
// fichier local
$xml = Xml::build('/home/awesome/unicorns.xml');

// fichier distant
$xml = Xml::build('http://bakery.cakephp.org/articles/rss');
```

Vous pouvez aussi construire des objets Xml en utilisant un tableau :

```
$data = array(
    'post' => array(
        'id' => 1,
        'title' => 'Best post',
        'body' => ' ... '
    )
);
$xml = Xml::build($data);
```

Si votre entrée est invalide, la classe Xml enverra une Exception :

```
$xmlString = 'What is XML?'
try {
    $xmlObject = Xml::build($xmlString); // Ici enverra une Exception
} catch (XmlException $e) {
    throw new InternalErrorException();
}
```

Note : DOMDocument (http ://php.net/domdocument) et SimpleXML (http ://php.net/simplexml) implement different API's. Assurez vous d'utiliser les bonnes méthodes sur l'objet que vous requêtez à partir d'un Xml.

Transformer une chaîne de caractères XML en tableau

Convertir des chaînes XML en tableaux est aussi facile avec la classe Xml. Par défaut, vous obtiendrez un objet SimpleXml en retour :

```
//Vieille méthode:
$xmlString = '<?xml version="1.0"?><root><child>value</child></root>';
$xmlObject = new Xml($xmlString);
$xmlArray = $xmlObject->toArray();

// Nouvelle méthode:
$xmlString = '<?xml version="1.0"?><root><child>value</child></root>';
$xmlArray = Xml::toArray(Xml::build($xmlString));
```

Si votre XML est invalide, cela enverra une Exception.

Transformer un tableau en une chaîne de caractères XML

```
// Vieille méthode:
$xmlArray = array('root' => array('child' => 'value'));
$xmlObject = new Xml($xmlArray, array('format' => 'tags'));
$xmlString = $xmlObject->toString();

// Nouvelle méthode:
```

```
$xmlArray = array('root' => array('child' => 'value'));
$xmlObject = Xml::fromArray($xmlArray, array('format' => 'tags')); // You can use Xml::bui
$xmlString = $xmlObject->asXML();
```

Votre tableau ne doit avoir qu'un élément de "niveau supérieur" et il ne doit pas être numérique. Si le tableau n'est pas dans le bon format, Xml va lancer une Exception. Des Exemples de tableaux invalides :

```
// Niveau supérieur avec une clé numérique
array(
    array('key' => 'value')
);

// Plusieurs clés au niveau supérieur
array(
    'key1' => 'première valeur',
    'key2' => 'autre valeur'
);
```

Warning : L'option format par défault a été changée de *attributes* pour *tags*. Cela a été fait pour rendre le Xml que la classe Xml génère plus compatible avec le Xml dans la nature. Attention si vous dépendez de celui-ci. Dans la nouvelle version, vous pouvez créer un tableau mixte avec des tags, des attributs et valeurs, utilisez juste le format en tags (ou ne dîtes rien, car c'est la valeur par défaut) et les clés préfixées qui sont sensées être des attributs avec @. Pour une valeur texte, mettez la clé à @.

```
$xmlArray = array(
    'projet' => array(
        '@id' => 1,
        'name' => 'Nom du projet, en tag',
        '@' => 'Valeur du projet'
    )
);
$xmlObject = Xml::fromArray($xmlArray);
$xmlString = $xmlObject->asXML();
```

Le contenu de `$xmlString` sera :

```
<?xml version="1.0"?>
<project id="1">Valeur du projet<name>Nom du projet, en tag</name></project>
```

Note : La structure des tableaux a été changée. Maintenant l'enfant doit avoir un sous-arbre et ne pas être dans le même arbre. En plus, les chaînes de caractères ne seront pas changées par `Inflector`. Regardez l'exemple ci-dessous :

```
$oldArray = array(
    'Projets' => array(
        array(
            'Projet' => array('id' => 1, 'title' => 'Projet 1'),
            'Industry' => array('id' => 1, 'name' => 'Industry 1')
        ),
```

```
        array(
            'Projet' => array('id' => 2, 'title' => 'Projet 2'),
            'Industry' => array('id' => 2, 'name' => 'Industry 2')
        )
    )
);

$newArray = array(
    'projets' => array(
        'projet' => array(
            array(
                'id' => 1, 'title' => 'Projet 1',
                'industry' => array('id' => 1, 'name' => 'Industry 1')
            ),
            array(
                'id' => 2, 'title' => 'Projet 2',
                'industry' => array('id' => 2, 'name' => 'Industry 2')
            )
        )
    )
);
```

Les deux engendreront le XML ci-dessous :

```
<?xml version="1.0"?>
<projets>
    <projet>
        <id>1</id>
        <title>Projet 1</title>
        <industry>
            <id>1</id>
            <name>Industry 1</name>
        </industry>
    </projet>
    <projet>
        <id>2</id>
        <title>Projet 2</title>
        <industry>
            <id>2</id>
            <name>Industry 2</name>
        </industry>
    </projet>
</projets>
```

Utiliser des Namespaces Pour utiliser les Namespaces XML, dans votre tableau vous devez créer une clé avec le nom `xmlns:` vers un namespace générique ou avec le préfixe `xmlns:` dans un namespace personnalisé. Regardez les exemples :

```
$xmlArray = array(
    'root' => array(
        'xmlns:' => 'http://cakephp.org',
        'child' => 'value'
```

```
    )
);
$xml1 = Xml::fromArray($xmlArray);

$xmlArray(
    'root' => array(
        'tag' => array(
            'xmlns:pref' => 'http://cakephp.org',
            'pref:item' => array(
                'item 1',
                'item 2'
            )
        )
    )
);
$xml2 = Xml::fromArray($xmlArray);
```

La valeur de `$xml1` et `$xml2` sera, respectivement :

```
<?xml version="1.0"?>
<root xmlns="http://cakephp.org"><child>value</child>


<?xml version="1.0"?>
<root><tag xmlns:pref="http://cakephp.org"><pref:item>item 1</pref:item><pref:item>item 2<
```

Créer un enfant La classe Xml de CakePHP 2.0 ne fournit pas la manipulation du contenu, cela doit être fait en utilisant SimpleXMLElement ou DOMDocument. Mais, comme CakePHP est trop sympa, ci-dessous vous avez les étapes pour créer un noeud enfant :

```
// CakePHP 1.3
$myXmlOriginal = '<?xml version="1.0"?><root><child>value</child></root>';
$xml = new Xml($myXmlOriginal, array('format' => 'tags'));
$xml->children[0]->createNode('young', 'new value');

// CakePHP 2.0 - En utilisant SimpleXML
$myXmlOriginal = '<?xml version="1.0"?><root><child>value</child></root>';
$xml = Xml::build($myXmlOriginal);
$xml->root->addChild('young', 'new value');

// CakePHP 2.0 - En utilisant DOMDocument
$myXmlOriginal = '<?xml version="1.0"?><root><child>value</child></root>';
$xml = Xml::build($myXmlOriginal, array('return' => 'domdocument'));
$child = $xml->createElement('young', 'new value');
$xml->firstChild->appendChild($child);
```

Astuce : Après avoir manipulé votre XML en utilisant SimpleXMLElement ou DomDocument vous pouvez utiliser `Xml::toArray()` sans problèmes.

API de Xml

Une classe usine de conversion pour créer des objets SimpleXML ou DOMDocument à partir d'un certain nombre de sources, y compris des chaînes, des tableaux et des urls distantes.

static `Xml::build`(*$input*, *$options = array()*)

Initialisez SimpleXMLElement ou DOMDocument à partir d'une chaîne de caractère XML donnée, d'un chemin de fichier, d'une URL ou d'un tableau.

Construire du XML à partir d'une chaîne de caractères :

```
$xml = Xml::build('<example>text</example>');
```

Construire du XML à partir d'une chaîne de caractères (sortie DOMDocument) :

```
$xml = Xml::build('<example>text</example>', array('return' => 'domdocument'));
```

Construire du XML à partir d'un chemin de fichier :

```
$xml = Xml::build('/path/to/an/xml/file.xml');
```

Construire à partir d'une URL distante :

```
$xml = Xml::build('http://example.com/example.xml');
```

Construire à partir d'un tableau :

```
$value = array(
    'tags' => array(
        'tag' => array(
            array(
                'id' => '1',
                'name' => 'defect'
            ),
            array(
                'id' => '2',
                'name' => 'enhancement'
        )
        )
    )
);
$xml = Xml::build($value);
```

Quand on construit du XML à partir d'un tableau, assurez-vous qu'il n'y a qu'un seul élément de niveau supérieur.

static `Xml::toArray`(*$obj*)

Convertit soit un objet SimpleXml, soit DOMDocument en un tableau.

CHAPITRE 8

Plugins

CakePHP vous permet de mettre en place une combinaison de controllers, models et vues et de les distribuer comme un plugin d'application packagé que d'autres peuvent utiliser dans leurs applications CakePHP. Vous avez un module de gestion des utilisateurs sympa, un simple blog, ou un module de service web dans une de vos applications ? Packagez le en plugin CakePHP afin de pouvoir la mettre dans d'autres applications.

Le principal lien entre un plugin et l'application dans laquelle il a été installé, est la configuration de l'application (connexion à la base de données, etc.). Autrement, il fonctionne dans son propre espace, se comportant comme il l'aurait fait si il était une application à part entière.

Installer un Plugin

Pour installer un plugin, commencez par simplement déplacer le dossier du du plugin dans votre dossier app/Plugin. Si vous installez un plugin nommé 'ContactManager' alors vous devez avoir un dossier dans app/Plugin appelé 'ContactManager' dans lequel vous aurez les Vues, les Models, les Controllers, webroot et tout autre répertoire du Plugin.

Nouveau dans CakePHP 2.0, les plugins ont besoin d'être chargés manuellement dans app/Config/bootstrap.php.

Vous pouvez soit les charger un par un, soit tous d'un coup dans un seul appel :

```
CakePlugin::loadAll(); // Charge tous les plugins d'un coup.
CakePlugin::load('ContactManager'); //Charge un seul plugin.
```

loadAll charge tous les plugins disponibles, bien que vous autorisant à configurer certains paramètres pour des plugins spécifiques. `load()` fonctionne de la même manière, mais charge seulement les plugins que vous avez explicitement spécifiés.

Configuration du Plugin

Vous pouvez faire beaucoup de choses avec les méthodes load et loadAll pour vous aider avec la configuration et le routing d'un plugin. Peut-être souhaiterez vous charger tous les plugins automatiquement, en spécifiant des routes et des fichiers de bootstrap pour certains plugins.

Pas de problèmes :

```
CakePlugin::loadAll(array(
    'Blog' => array('routes' => true),
    'ContactManager' => array('bootstrap' => true),
    'WebmasterTools' => array('bootstrap' => true, 'routes' => true),
));
```

Avec ce type de configuration, vous n'avez plus besoin d'include() ou de require() manuellement une configuration de plugin ou un fichier de routes–Cela arrive automatiquement au bon moment et à la bonne place. Un paramètre totalement identique peut avoir été fourni à la méthode load(), ce qui aurait chargé seulement ces trois plugins, et pas le reste.

Au final, vous pouvez aussi spécifier un ensemble de valeurs dans defaults pour loadAll qui s'applique à chaque plugin qui n'a pas de configuration spécifique.

Chargez le fichier bootstrap à partir de tous les plugins, et les routes à partir du plugin Blog :

```
CakePlugin::loadAll(array(
    array('bootstrap' => true),
    'Blog' => array('routes' => true)
));
```

Notez que tous les fichiers spécifiés doivent réellement exister dans le(s) plugin(s) configurés ou PHP vous donnera des avertissements pour chaque fichier qu'il ne peut pas charger. C'est particulièrement important à retenir quand on spécifie defaults pour tous les plugins.

Certains plugins ont besoin en supplément de créer une ou plusieurs tables dans votre base de données. Dans ces cas, ils incluent souvent un fichier de schéma que vous appelez à partir du shell de cake comme ceci :

```
user@host $ cake schema create --plugin ContactManager
```

La plupart des plugins indiqueront dans leur documentation leur propre procédure pour les configurer et configurer la base de données. Certains plugins nécessiteront plus de configuration que d'autres.

Aller plus loin avec le bootstrapping

Si vous souhaitez charger plus d'un fichier bootstrap pour un plugin. Vous pouvez spécifier un tableau de fichiers avec la clé de configuration bootstrap :

```
CakePlugin::loadAll(array(
    'Blog' => array(
        'bootstrap' => array(
            'config1',
```

```
            'config2'
        )
    )
));
```

Vous pouvez aussi spécifier une fonction qui pourra être appelée quand le plugin est chargé :

```
function aCallableFunction($pluginName, $config) {

}

CakePlugin::loadAll(array(
    'Blog' => array(
        'bootstrap' => 'aCallableFunction'
    )
));
```

Utiliser un Plugin

Vous pouvez référencer les controllers, models, components, behaviors et helpers du plugin en préfixant le nom du plugin avant le nom de classe.

Par exemple, disons que vous voulez utiliser le ContactInfoHelper du plugin ContactManager pour sortir de bonnes informations de contact dans une de vos vues. Dans votre controller, le tableau $helpers pourrait ressembler à ceci :

```
public $helpers = array('ContactManager.ContactInfo');
```

Vous serez ensuite capable d'accéder à ContactInfoHelper comme tout autre helper dans votre vue, comme ceci :

```
echo $this->ContactInfo->address($contact);
```

Créer Vos Propres Plugins

En exemple de travail, commençons par créer le plugin ContactManager référencé ci-dessus. Pour commencer, nous allons configurer votre structure de répertoire basique. Cela devrait ressembler à ceci :

```
/app
    /Plugin
        /ContactManager
            /Controller
                /Component
            /Model
                /Behavior
            /View
                /Helper
                /Layouts
```

Notez que le nom du dossier du plugin, '**ContactManager**'. Il est important que ce dossier ait le même nom que le plugin.

Dans le dossier plugin, vous remarquerez qu'il ressemble beaucoup à une application CakePHP, et c'est au fond ce que c'est. Vous n'avez à inclure aucun de vos dossiers si vous ne les utilisez pas. Certains plugins peuvent ne contenir qu'un Component ou un Behavior, et dans certains cas, ils peuvent carrément ne pas avoir de répertoire 'View'.

Un plugin peut aussi avoir tous les autres répertoires que votre application a, comme Config, Console, Lib, webroot, etc...

Note : Si vous voulez être capable d'accéder à votre plugin avec une URL, vous devrez définir un AppController et un AppModel pour le plugin. Ces deux classes spéciales sont nommées d'après le plugin, et étendent les AppController et AppModel de notre application parente. Voilà à quoi cela devrait ressembler pour notre exemple de ContactManager :

```
// /app/Plugin/ContactManager/Controller/ContactManagerAppController.php:
class ContactManagerAppController extends AppController {
}
```

```
// /app/Plugin/ContactManager/Model/ContactManagerAppModel.php:
class ContactManagerAppModel extends AppModel {
}
```

Si vous oubliez de définir ces classes spéciales, CakePHP vous donnera des erreurs "Missing Controller" jusqu'à ce que ce soit fait.

Merci de noter que le processus de création de plugins peut être méchamment simplifié en utilisant le shell de Cake.

Pour cuisiner un plugin, merci d'utiliser la commande suivante :

```
user@host $ cake bake plugin ContactManager
```

Maintenant vous pouvez cuisiner en utilisant les mêmes conventions qui s'appliquent au reste de votre app. Par exemple - baking controllers :

```
user@host $ cake bake controller Contacts --plugin ContactManager
```

Merci de vous référer au chapitre *Génération de code avec Bake* si vous avez le moindre problème avec l'utilisation de la ligne de commande.

Controllers du Plugin

Les controllers pour notre plugin ContactManager seront stockés dans /app/Plugin/ContactManager/Controller/. Puisque la principale chose que nous souhaitons faire est la gestion des contacts, nous aurons besoin de créer un ContactsController pour ce plugin.

Ainsi, nous mettons notre nouveau ContactsController dans /app/Plugin/ContactManager/Controller et il ressemblerait à cela :

```
// app/Plugin/ContactManager/Controller/ContactsController.php
class ContactsController extends ContactManagerAppController {
    public $uses = array('ContactManager.Contact');

    public function index() {
        //...
    }
}
```

Note : Ce controller étend AppController du plugin (appelé ContactManagerAppController) plutôt que l'AppController de l'application parente.

Notez aussi comment le nom du model est préfixé avec le nom du plugin. C'est nécessaire pour faire la différence entre les models dans les plugins et les models dans l'application principale.

Dans ce cas, le tableau $uses ne serait pas nécessaire comme dans ContactManager. Contact sera le model par défaut pour ce controller, cependant, il est inclu pour démontrer comment faire préceder proprement le nom du plugin.

Si vous souhaitez accéder à ce que nous avons obtenu jusqu'à présent, visitez /contact_manager/contacts. Vous devriez obtenir une erreur "Missing Model" parce que nous n'avons pas un model Contact déjà défini.

Models du Plugin

Les Models pour le plugin sont stockés dans /app/Plugin/ContactManager/Model. Nous avons déjà défini un ContactsController pour ce plugin, donc créons le models pour ce controller, appelé Contact :

```
// /app/Plugin/ContactManager/Model/Contact.php:
class Contact extends ContactManagerAppModel {
}
```

Visiter /contact_manager/contacts maintenant (Etant donné, que vous avez une table dans votre base de données appelée 'contacts') devrait nous donner une erreur "Missing View". Créons la ensuite.

Note : Si vous avez besoin de réferencer un model dans votre plugin, vous avez besoin d'inclure le nom du plugin avec le nom du model, séparé d'un point.

Par exemple :

```
// /app/Plugin/ContactManager/Model/Contact.php:
class Contact extends ContactManagerAppModel {
    public $hasMany = array('ContactManager.AltName');
}
```

Si vous préférez que les clés du tableau pour l'association n'aient pas le préfixe du plugin sur eux, utilisez la syntaxe alternative :

```
// /app/Plugin/ContactManager/Model/Contact.php:
class Contact extends ContactManagerAppModel {
    public $hasMany = array(
        'AltName' => array(
            'className' => 'ContactManager.AltName'
        )
    );
}
```

Vues du Plugin

Les Vues se comportent exactement comme elles le font dans les applications normales. Placez les juste dans le bon dossier à l'intérieur du dossier /app/Plugin/[PluginName]/View/. Pour notre plugin ContactManager, nous aurons besoin d'une vue pour notre action ContactsController : :index(), ainsi incluons ceci aussi :

```
// /app/Plugin/ContactManager/View/Contacts/index.ctp:
<h1>Contacts</h1>
<p>Ce qui suit est une liste triable de vos contacts</p>
<!-- Une liste triable de contacts irait ici....-->
```

Note : Pour des informations sur la façon d'utiliser les éléments à partir d'un plugin, regardez *Elements*.

Redéfinition des vues de plugin à partir de l'intérieur de votre application

Vous pouvez redéfinir toutes les vues du plugin à partir de l'intérieur de votre app en utilisant des chemins spéciaux. Si vous avez un plugin appelé 'ContactManager', vous pouvez redéfinir les fichiers de vue du plugin avec une logique de vue de l'application plus spécifique, en créant des fichiers en utilisant le template suivant "app/View/Plugin/[Plugin]/[Controller]/[view].ctp". Pour le controller Contacts, vous pouvez faire le fichier suivant :

```
/app/View/Plugin/ContactManager/Contacts/index.ctp
```

Créer ce fichier vous permettra de redéfinir "/app/Plugin/ContactManager/View/Contacts/index.ctp".

Plugin assets

Les assets web du plugin (mais pas les fichiers de PHP) peuvent être servis à travers le répertoire 'webroot' du plugin, juste comme les assets de l'application principale :

```
app/Plugin/ContactManager/webroot/
                                    css/
                                    js/
                                    img/
                                    flash/
                                    pdf/
```

Vous pouvez mettre tout type de fichier dans tout répertoire, juste comme un webroot habituel. La seule restriction est que `MediaView` a besoin de connaître le mime-type de cet asset.

Mais garder à l'esprit que la gestion des assets statiques, comme les images, le Javascript et les fichiers CSS des plugins à travers le Dispatcher est incroyablement innéficace. Il est grandement recommandé de les symlinker pour la production. Par exemple comme ceci :

```
ln -s app/Plugin/YourPlugin/webroot/css/yourplugin.css app/webroot/css/yourplugin.css
```

Lier aux plugins

Faîtes précéder simplement /nom_plugin/ pour le début d'une requête pour un asset dans ce plugin, et cela fonctionnera comme si l'asset était dans le webroot de votre application.

Par exemple, lier le '/contact_manager/js/some_file.js' servirait l'asset 'app/Plugin/ContactManager/webroot/js/some_file.js'.

Note : Il est important de noter que le préfixe de **/votre_plugin/** avant le chemin de asset. Et la magie opére !

Modifié dans la version 2.1 : Utilisez *plugin syntax* pour accéder aux assets. Par exemple dans votre View : < ?php echo $this->Html->css("ContactManager.style") ; ?>

Components, Helpers et Behaviors

Un plugin peut avoir des Components, Helpers et Behaviors tout comme un une appplication CakePHP classique. Vous pouvez soit créer des plugins qui sont composés seulement de Components, Helpers ou Behaviors qui peuvent être une bonne façon de construire des Components réutilisables qui peuvent être facilement déplacés dans tout projet.

Construire ces components est exactement le même chose que de les construire à l'intérieur d'une application habituelle, avec aucune convention spéciale de nommage.

Faire référence avec votre component, depuis l'intérieur ou l'extérieur de votre plugin nécessite seulement que le préfixe du nom du plugin avant le nom du component. Par exemple :

```
// Component défini dans le plugin 'ContactManager'
class ExampleComponent extends Component {
}

// dans vos controllers:
public $components = array('ContactManager.Exemple');
```

La même technique s'applique aux Helpers et aux Behaviors.

Note : A la création de Helpers, vous verrez que AppHelper n'est pas automatiquement disponible. Vous pouvez déclarer les ressources dont vous avez besoin avec les uses :

```
// Déclarez le use de AppHelper pour le Helper de votre Plugin
App::uses('AppHelper', 'View/Helper');
```

Etendez votre Plugin

Cet exemple est un bon début pour un plugin, mais il y a beaucoup plus à faire. En règle général, tout ce que vous pouvez faire avec votre application, vous pouvez le faire à l'intérieur d'un plugin à la place.

Continuez, incluez certaines librairies tierces dans 'Vendor', ajoutez des nouveaux shells à la console de cake, et n'oubliez pas de créez des cas de test ainsi les utilisateurs de votre plugin peuvent automatiquement tester les fonctionnalités de votre plugin !

Dans notre exemple ContactManager, nous pourrions créer des actions add/remove/edit/delete dans le ContactsController, intégrez la validation dans le model Contact, et intégrez la fonctionnalité à laquelle on pourrait s'attendre quand on gère ses contacts. A vous de décider ce qu'il fait intégrer dans vos plugins. N'oubliez juste pas de partager votre code avec la communauté afin que tout le monde puisse bénéficier de votre component génial et réutilisable !

Astuces pour les Plugins

Une fois qu'un plugin a été installé dans /app/Plugin, vous pouvez y accéder à l'URL /nom_plugin/nom_controller/action. Dans notre exemple de plugin ContactManager, nous accédons à notre ContactsController à l'adresse /contact_manager/contacts.

Quelques astuces de fin lorque l'on travaille avec les plugins dans vos applications CakePHP :

- Si vous n'avez pas un [Plugin]AppController et [Plugin]AppModel, vous aurez des erreurs de type get missing Controller lorsque vous essayez d'accéder à un controller d'un plugin.
- Vous pouvez définir vos propres layouts pour les plugins, dans le dossier de app/Plugin/[Plugin]/View/Layouts. Sinon, les plugins utiliseront les layouts du dossier /app/View/Layouts par défaut.
- Vous pouvez établir une communication inter-plugin en utilisant `$this->requestAction('/plugin_name/controller_name/action');` dans vos controllers.
- Si vous utilisez requestAction, assurez-vous que les noms des controllers et des models sont aussi uniques que possibles. Sinon, vous aurez des erreurs PHP de type "redefined class ...".

CHAPITRE 9

Console et Shells

CakePHP ne dispose pas seulement d'un framework web mais aussi d'une console de framework pour la création d'applications. Les applications par console sont idéales pour la gestion d'une variété de tâches d'arrière-plan comme la maintenance et l'achèvement du travail en-dehors du cycle de requête-réponse. Les applications par la console CakePHP vous permettent de réutilise les classes de votre application à partir de lignes de commande.

CakePHP dispose d'un certain nombre d'applications fournies pour la console. Certaines de ces applications sont utilisées de concert avec les fonctionnalités de CakePHP (comme ACL ou i18n), et d'autres sont pour une utilisation générale pour que votre travail se fasse plus vite.

La console de CakePHP

Cette section fournit une introduction sur la ligne de commande dans CakePHP. Si vous avez besoin d'accéder à vos classes MVC de CakePHP dans une tâche cron ou tout autre script de ligne de commande, cette section est pour vous.

PHP fournit un puissant client CLI qui rend l'interfaçage avec votre système de fichier et vos applications plus facile. La console CakePHP fournit un framework de création de scripts shell. La console utilise un ensemble de répartiteur de types pour charger un shell ou une tâche, et lui passer des paramètres.

Note : Une installation de PHP contruite avec la ligne de commande (CLI) doit être disponible sur le système où vous prévoyez d'utiliser la console.

Avant d'entrer dans les spécificités, assurons-nous que vous pouvez exécuter la console CakePHP. Tout d'abord, vous devrez ouvrir un shell système. Les exemples présentés dans cette section sont issus du bash, mais la console CakePHP est également compatible Windows. Exécutons le programme Console depuis le bash. Cet exemple suppose que l'utilisateur est actuellement connecté dans l'invite bash et qu'il est root sur une installation CakePHP.

Les applications CakePHP contiennent un répertoire `Console` qui contient tous les shells et les tâches pour une application. Il est aussi livré avec un exécutable :

```
$ cd /path/to/cakephp/app
$ Console/cake
```

Mais il est préférable d'ajouter l'exécutable du coeur de cake dans votre système de path afin que vous puissiez utiliser la commande cake de partout. C'est plus pratique quand vous créez de nouveaux projets. Regardez *Ajouter cake à votre système path* pour voir la façon de rendre `cake` disponible dans tout le système.

Lancez la Console avec aucun argument entraîne ce message d'aide :

```
Welcome to CakePHP v2.0.0 Console
---------------------------------------------------------------
App : app
Path: /path/to/cakephp/app/
---------------------------------------------------------------
Current Paths:

 -app: app
 -working: /path/to/cakephp/app
 -root: /path/to/cakephp/
 -core: /path/to/cakephp/core

Changing Paths:

your working path should be the same as your application path
to change your path use the '-app' param.
Example: -app relative/path/to/cakephp/app or -app /absolute/path/to/cakephp/app

Available Shells:

 acl [CORE]                                i18n [CORE]
 api [CORE]                                import [app]
 bake [CORE]                               schema [CORE]
 command_list [CORE]                       testsuite [CORE]
 console [CORE]                            upgrade [CORE]

To run a command, type 'cake shell_name [args]'
To get help on a specific command, type 'cake shell_name help'
```

La première information affichée est en rapport avec les chemins. Ceci est particulièrement pratique si vous exécutez la Console depuis différents endroits de votre système de fichier.

Beaucoup d'utilisateurs ajoutent la console CakePHP à leur path système afin qu'elle puisse être facilement accessible. L'affichage des chemins de workdir, root, app et corevous permet de voir où la Console fera des changements. Pour changer le dossier app par celui dans lequel vous souhaitez travailler, vous pouvez fournir son chemin comme premier argument de la ligne de commande cake. L'exemple suivant montre comment spécifier un dossier app, en supposant que vous avez déjà ajouté le dossier de la console à votre `PATH`

```
$ cake -app /path/to/cakephp/app
```

Le chemin fourni peut être relatif au répertoire courant ou fourni sous forme de chemin absolu.

Ajouter cake à votre système path

Si vous êtes sur un système *nix (linux, MacOSX), les étapes suivantes vous permettront de rendre cake executable dans votre système path.

1. Localisez où se trouve votre installation de cakephp et le cake executable. Par exemple `/Users/mark/cakephp/lib/Cake/Console/cake`
2. Modifiez votre fichier `.bashrc` ou `.bash_profile` dans votre répertoire home, et ajoutez ce qui suit :

```
export PATH="$PATH:/Users/mark/cakephp/lib/Cake/Console"
```

3. Rechargez la configuration bash ou ouvrez un nouveau terminal, et `cake` devrait fonctionner n'importe où.

Si vous êtes sur Windows Vista ou 7, vous devrez suivre les étapes suivantes.

1. Localisez l'emplacement où se trouvent votre installation CakePHP et l'executable cake. Par exemple `C:\xampp\htdocs\cakephp\lib\Cake\Console`
2. Ouvrez la fenêtre des propriétés du Système de votre ordinateur. Vous essaierez le raccourci clavier Windows Key + Pause ou Windows Key + Break. Ou, à partir du Bureau, faîtes un click droit sur Mon Ordinateur, clickez sur Propriétés et clickez sur le lien Paramètres avancés du système dans la colonne de gauche.
3. Allez sous l'onglet Avancé et clickez sur le bouton des Variables d'Environnement.
4. Dans la portion des Variables Sytèmes, cherchez le chemin de la variable et double-clickez dessus pour la modifier.
5. Ajoutez le chemin de l'installation de `cake` suivi par un point virgule. On pourrait avoir par exemple :

```
%SystemRoot%\system32;%SystemRoot%;C:\xampp\htdocs\cakephp\lib\Cake\Console;
```

6. Clickez Ok et `cake` devrait fonctionner partout.

Créer un shell

Créons un shell pour l'utilisation dans la Console. Pour cet exemple, nous créerons un simple shell Hello world. Dans le répertoire `Console/Command` de votre application, créez `HelloShell.php`. Mettez le code suivant dedans :

```
class HelloShell extends AppShell {
    public function main() {
        $this->out('Hello world.');
    }
}
```

Les conventions pour les classes de shell sont que les noms de classe doivent correspondre au nom du fichier, avec Shell en suffixe. Dans notre shell, nous avons crée une méthode `main()`. Cette méthode est appelée quand un shell est appelé avec aucune commande supplémentaire. Nous allons ajouter quelques commandes en plus dans un moment, mais pour l'instant lançons juste notre shell. Depuis le répertoire de votre application, lancez :

```
Console/cake hello
```

Vous devriez voir la sortie suivante :

```
Welcome to CakePHP v2.0.0 Console
---------------------------------------------------------------
App : app
Path: /Users/markstory/Sites/cake_dev/app/
---------------------------------------------------------------
Hello world.
```

Comme mentionné avant, la méthode `main()` dans les shells est une méthode spéciale appelée tant qu'il n'y a pas d'autres commandes ou arguments donnés au shell. Vous pouvez aussi remarquer que HelloShell étend `AppShell`. Un peu comme *Le Controller App*, AppShell vous donne une classe de base pour contenir toutes les fonctions ordinaires ou logiques. Vous pouvez définir un AppShell en créant `app/Console/Command/AppShell.php`. Si vous n'en avez pas un, CakePHP en utilisera une integrée. Comme notre méthode principale n'était pas très intéressante, ajoutons une autre commande qui fait quelque chose :

```
class HelloShell extends AppShell {
    public function main() {
        $this->out('Hello world.');
    }

    public function hey_there() {
        $this->out('Hey there ' . $this->args[0]);
    }
}
```

Après avois sauvegardé ce fichier, vous devriez être capable de lancer `Console/cake hello hey_there your-name` et de voir vos noms affichés. Toute méthode publique non préfixée par un _ peut être appelée à partir de ligne de commande. Dans notre méthode `hey_there`, nous utilisons aussi `$this->args`, cette propriété contient un tableau de tous les arguments de position fournis à une commande. Vous pouvez aussi utiliser des switches ou des options sur les shells des applications, ils sont disponibles dans la variable `$this->params`, mais nous verrons ça bientôt.

Lorsque vous utilisez la méthode `main()`, vous n'êtes pas capable d'utiliser les arguments de position ou les paramètres. Cela parce que le premier argument de position ou l'option est interprétée en tant que nom de commande. Si vous voulez utiliser des arguments et des options, vous devriez utiliser un autre nom de méthode que `main`.

Utiliser les Models dans vos shells

Vous avez souvent besoin d'accéder à la logique métier de votre application dans les utilitaires de shell. CakePHP rend cela super facile. En configurant une propriété `$uses`, vous pouvez définir un tableau de

models auxquels vous voulez accéder dans votre shell. Les models définis sont chargés en propriétés attachées à votre shell, juste comme un controller obtient les models qui lui sont attachés :

```
class UserShell extends AppShell {
    public $uses = array('User');

    public function show() {
        $user = $this->User->findByUsername($this->args[0]);
        $this->out(print_r($user, true));
    }
}
```

Le shell ci-dessus récupérera un utilisateur par son username et affichera l'information stockée dans la base de données.

Les tâches Shell

Il y aura des fois où quand vous construirez des applications plus poussées via la console, vous voudrez composer des fonctionnalités dans des classes réutilisables qui peuvent être partagées à travers plusieurs shells. Les tâches vous permettent d'extraire des commandes dans des classes. Par exemple, `bake` est fait entièrement de tâches. Vous définissez les tâches d'un shell en utilisant la propriété `$tasks` :

```
class UserShell extends AppShell {
    public $tasks = array('Template');
}
```

Vous pouvez utiliser les tâches à partir de plugins en utilisant la *plugin syntax* standard. Les tâches sont stockées dans `Console/Command/Task/` dans les fichiers nommées d'après leur classe. Ainsi si vous étiez sur le point de créer une nouvelle tâche 'FileGenerator', vous pourriez créer `Console/Command/Task/FileGeneratorTask.php`.

Chaque tâche doit au moins intégrer une méthode `execute()`. Le ShellDispatcher appelera cette méthode quand la tâche est invoquée. une classe de tâche ressemble à cela :

```
class FileGeneratorTask extends Shell {
    public $uses = array('User');
    public function execute() {

    }
}
```

Un shell peut aussi accéder à ses tâches en tant que propriétés, ce qui rend les tâches meilleures pour la réutilisation de focntions identiques à *Components (Composants)* :

```
// trouvé dans Console/Command/SeaShell.php
class SeaShell extends AppShell {
    public $tasks = array('Sound'); // trouvé dans Console/Command/Task/SoundTask.php
    public function main() {
        $this->Sound->execute();
    }
}
```

Vous pouvez aussi accéder aux tâches directement à partir de la ligne de commande :

```
$ cake sea sound
```

Note : Afin d'accéder aux tâches directement à partir de ligne de commande, la tâche **doit** être inclue dans la propriété $tasks de la classe shell. Pour ce faire, soyez averti qu'une méthode appelée "sound" dans la classe SeaShell redéfinira la capacité d'accès à la fonctionnalité dans la tâche Sound spécifiée dans le tableau $tasks.

Chargement à la volée des tâches avec TaskCollection

Vous pouvez charger les tâches à la volée en utilisant l'objet Task Collection. Vous pouvez charger les tâches qui ne sont pas déclarées dans $tasks de cette façon :

```
$Project = $this->Tasks->load('Project');
```

Chargera et retournera une instance ProjectTask. Vous pouvez charger les tâches à partir des plugins en utilisant :

```
$ProgressBar = $this->Tasks->load('ProgressBar.ProgressBar');
```

Invoquer d'autres shells à partir de votre shell

Les Shells n'ont plus directement accès à ShellDispatcher, à travers *$this->Dispatch*. Il y a cependant beaucoup de cas où vous voulez invoquer un shell à partir d'un autre. *Shell : :dispatchShell()* vous donne la possibilité d'appeler d'autres shells en fournissant le *argv* pour le shell sub. Vous pouvez fournir des arguments et des options soit en variables args ou en chaînes de caractères :

```
// En chaînes de caractère
$this->dispatchShell('schema create Blog --plugin Blog');

// En tableau
$this->dispatchShell('schema', 'create',  'Blog', '--plugin',  'Blog');
```

Ce qui est au-dessus montre comment vous pouvez appeler le shell schema pour un plugin à partir du shell de votre plugin.

Niveaux de sortie de la Console

Les Shells ont souvent besoin de différents niveaux de verbosité. Quand vous lancez une tâche cron, la plupart des sorties ne sont pas nécessaires. Et il y a des fois où vous n'êtes pas interessés dans tout ce qu'un shell a à dire. Vous pouvez utiliser des niveaux de sortie pour signaler la sortie de façon appropriée. L'utilisateur du shell peut ensuite décider pour quel niveau de détail ils sont interessés en configurant le bon flag quand on appelle le shell. `Shell::out()` supporte 3 types de sortie par défaut.

– QUIET - Seulement des informations importantes doivent être marquées pour une paisible.
– NORMAL - Le niveau par défaut, et un usage normal
– VERBOSE - Les messages marqués qui peuvent être trop ennuyeux pour une utilisation quotidienne, mais aide au debugging en VERBOSE

Vous pouvez marquer la sortie comme suit :

```
// apparaitra à tous les niveaux.
$this->out('Quiet message', 1, Shell::QUIET);

// n'apparaitra pas quand une sortie quiet est togglé
$this->out('message normal', 1, Shell::NORMAL);
$this->out('message loud', 1, Shell::VERBOSE);

// would only appear when verbose output is enabled.
$this->out('extra message', 1, Shell::VERBOSE);
```

Vous pouvez contrôler le niveau de sortie des shells, en utilisant les options `--quiet` et `--verbose`. Ces options sont ajoutées par défaut, et vous autorise à contrôler la cohérence du niveau de sortie à l'intérieur de vos shells CakePHP.

Style de sortie

La Style de sortie est fait en incluant les tags - juste comme le html - dans votre sortie.

ConsoleOutput remplacera ces tags avec la bonne séquence de code ansi, ou supprimera les tags si vous êtes sur une console qui ne supporte pas les codes ansi. Il y a plusieurs styles intégrés, et vous pouvez en créer plus. Ceux intégrés sont

– `error` Messages d'Erreur. Texte rouge souligné.
– `warning` Warning messages. Texte jaune.
– `info` Messages d'informations. Texte cyan.
– `comment` Texte supplémentaire. Texte bleu.
– `question` Texte qui est une question, ajouté automatiquement par shell.

Vous pouvez créer des styles supplémentaires en utilisant *$this->stdout->styles()*. Pour déclarer un nouveau style de sortie, vous pouvez faire :

```
$this->stdout->styles('flashy', array('text' => 'magenta', 'blink' => true));
```

Cela vous permettra d'utiliser un tag `<flashy>` dans la sortie de votre shell, et si les couleurs ansi sont activées, ce qui suit sera rendu en texte magenta clignotant `$this->out('<flashy>Whoooa</flashy> Quelque chose a posé problème');`.

Quand vous définissez les styles, vous pouvez utiliser les couleurs suivantes pour les attributs *text* et *background* :

– black
– red
– green
– yellow
– blue
– magenta
– cyan

– white
Vous pouvez aussi utiliser les options suivantes en commutateurs boléens, en les définissant à une valeur true qui les activent.
– bold
– underline
– blink
– reverse
Ajouter un style le rend aussi disponible pour toutes les instances de ConsoleOutput, donc vous n'avez pas à redeclarer les styles pour les deux objets stdout et stderr.

Enlever la coloration

Bien que la coloration soit vraiment géniale, il peut y avoir des fois où vous voulez l'arrêter, ou forcer à l'avoir :

```
$this->output->outputAs(ConsoleOutput::RAW);
```

Ce qui est au-dessus met la sortie objet dans un mode de sortie en ligne. Dans le mode de sortie en ligne, il n'y a aucun style du tout. Il y a trois modes que vous pouvez utiliser.
– `ConsoleOutput::RAW` - Sortie en ligne, aucun style ou format ne sera fait C'est un bon mode à utiliser si vous sortez du XML ou si voulez débugger pourquoi votre style ne marche pas.
– `ConsoleOutput::PLAIN` - Sortie en texte plein, les tags de style connus seront enlevés de la sortie.
– `ConsoleOutput::COLOR` - La sortie avec couleur enlève les codes en place.
Par défaut sur les systèmes *nix, les objets ConsoleOutput ont par défaut de la couleur. Sur les systèmes windows, la sortie simple est mise par défaut sauf si la variable d'environnement `ANSICON` est présente.

Configurer les options et générer de l'aide

class `ConsoleOptionParser`

Le parsing d'option de la Console dans CakePHP a toujours été un peu différent par rapport aux autres consoles en ligne de commande. Dans 2.0, `ConsoleOptionParser` aide à fournir un parser d'option et d'argument plus familier en ligne de commande.

OptionParsers vous permet d'accomplir deux buts en même temps. Premièrement, ils vous permettent de définir les options et arguments, séparant la validation basique des entrées et votre code. Deuxièmement, il vous permet de fournir de la documentation, qui est utilisé pour bien générer le fichier d'aide formaté.

La console du framework récupère votre parser d'option du shell en appelant `$this->getOptionParser()`. Surcharger cette méthode vous permet de configurer l'OptionParser pour faire correspondre les entrées attendues de votre shell. Vous pouvez aussi configurer les parsers d'option des sous-commandes, ce qui vous permet d'avoir des parsers d'option différents pour les sous-commandes et les tâches. ConsoleOptionParser implémente une interface courant et inclut les méthodes pour configurer facilement les multiple options/arguments en une fois. :

```
public function getOptionParser() {
    $parser = parent::getOptionParser();
    //configure parser
    return $parser;
}
```

Configurer un option parser avec l'interface courante

Toutes les méthodes utilisées pour configurer le parser peuvent être chainées, vous permettant de définir l'intégralité des options du parser en une unique série d'appel de méthodes :

```
public function getOptionParser() {
    $parser = parent::getOptionParser();
    $parser->addArgument('type', array(
        'help' => 'Either a full path or type of class.'
    ))->addArgument('className', array(
        'help' => 'A CakePHP core class name (e.g: Component, HtmlHelper).'
    ))->addOption('method', array(
        'short' => 'm',
        'help' => __('The specific method you want help on.')
    ))->description(__('Lookup doc block comments for classes in CakePHP.'));
    return $parser;
}
```

Les méthodes autorisant le chainage sont :
- description()
- epilog()
- command()
- addArgument()
- addArguments()
- addOption()
- addOptions()
- addSubcommand()
- addSubcommands()

`ConsoleOptionParser::description`(*$text = null*)

Récupère ou définit la description pour le parser d'option. La description affiche en haut l'argument et l'information d'option. En passant soit un tableau, soit une chaîne, vous pouvez définir la valeur de la description. Appeler sans arguments va retourner la valeur actuelle :

```
// Définit plusieurs lignes en une fois
$parser->description(array('line one', 'line two'));

// Lit la valeur actuelle
$parser->description()
```

`ConsoleOptionParser::epilog`(*$text = null*)

Récupère ou définit l'epilog pour le parser d'option. L'epilog est affichée après l'argument et l'information d'option. En passant un tableau ou une chaîne, vous pouvez définir la valeur de epilog. Appeler avec aucun argument va retourner la valeur actuelle :

```
// Définit plusieurs lignes en une fois
$parser->epilog(array('line one', 'line two'));

// Lit la valeur actuelle
$parser->epilog()
```

Ajouter des arguments

`ConsoleOptionParser::addArgument`(*$name*, *$params = array()*)

Les arguments de position sont fréquemment utilisés dans les outils en ligne de commande, et `ConsoleOptionParser` vous permet de définir les arguments de position ainsi que de les rendre obligatoires. Vous pouvez ajouter des arguments un à la fois avec `$parser->addArgument();` ou plusieurs à la fois avec `$parser->addArguments();` :

```
$parser->addArgument('model', array('help' => 'The model to bake'));
```

Vous pouvez utiliser les options suivantes lors de la création d'un argument :
- `help` Le texte d'aide à afficher pour cet argument.
- `required` Si le paramètre est obligatoire.
- **`index` L'index pour l'argument, si non défini à gauche, l'argument sera** mis à la fin des arguments. Si vous définissez le même index deux fois, la première option sera écrasée.
- **`choices` Un tableau de choix valides pour cet argument. Si vide à gauche,** toutes les valeurs sont valides. Une exception sera lancée quand parse() rencontre une valeur non valide.

Les arguments qui ont été marqués comme nécessaires vont lancer une exception lors du parsing de la commande si ils ont été omis. Donc vous n'avez pas à gérer cela dans votre shell.

`ConsoleOptionParser::addArguments`(*array $args*)

Si vous avez un tableau avec plusieurs arguments, vous pouvez utiliser `$parser->addArguments()` pour ajouter plusieurs arguments en une fois. :

```
$parser->addArguments(array(
    'node', array('help' => 'The node to create', 'required' => true),
    'parent' => array('help' => 'The parent node', 'required' => true)
));
```

Comme avec toutes les méthodes de construction avec ConsoleOptionParser, addArguments peuvent être utilisés comme des parties d'une chaîne de méthode courante.

Validation des arguments

Lors de la création d'arguments de position, vous pouvez utiliser le flag `required`, pour indiquer qu'un argument doit être présent quand un shell est appelé. De plus, vous pouvez utiliser `choices` pour forcer un argument pour qu'il soit une liste de choix valides :

```
$parser->addArgument('type', array(
    'help' => 'Le type de noeud avec lequel intéragir.',
    'required' => true,
    'choices' => array('aro', 'aco')
));
```

Ce qui est au-dessus va créer un argument qui est nécessaire et a une validation sur l'entrée. Si l'argument est soit manquant, soit a une valeur incorrecte, une exception sera levée et le shell sera arreté.

Ajouter des Options

`ConsoleOptionParser::addOption`(*$name*, *$options = array()*)

Les options ou les flags sont aussi fréquemment utilisés avec les outils de ligne de commande. `ConsoleOptionParser` supporte la création d'options avec les deux verbose et short aliases, fournissant les valeurs par défaut et créant des switches en boléen. Les options sont créées avec soit `$parser->addOption()`, soit `$parser->addOptions()`.:

```
$parser->addOption('connection', array(
    'short' => 'c'
    'help' => 'connection',
    'default' => 'default'
));
```

Ce qui est au-dessus vous permet l'utilisation soit d `cake myshell --connection=other`, soit `cake myshell --connection other`, ou soit `cake myshell -c other` lors de l'appel au shell. Vous pouvez aussi créer des switches de boléen, ces switches ne consomment pas de valeurs, et leur présence les active juste dans les paramètres parsés. :

```
$parser->addOption('no-commit', array('boolean' => true));
```

Avec cette option, lors de l'appel d'un shell comme `cake myshell --no-commit something` le paramètre no-commit aurait une valeur à true, et 'something' serait traité comme un argument de position. Les options intégrées `--help`, `--verbose`, et `--quiet` utilisent cette fonctionnalité.

Lors de la création d'options, vous pouvez utiliser les options suivantes pour définir le comportement de l'option :
- `short` - La variante de la lettre unique pour cette option, laissez à non définie pour n'en avoir aucun.
- `help` - Le texte d'aide pour cette option. Utilisé lors de la génération d'aide pour l'option.
- `default` - La valeur par défaut pour cette option. Si elle n'est pas définie, la valeur par défaut sera true.
- `boolean` - L'option n'utilise aucune valeur, c'est juste un switch de boléen. Par défaut à false.
- `choices` Un tableau de choix valides pour cette option. Si elle est vide, toutes les valeurs sont valides. Une exception sera lancée lorque parse() rencontre une valeur non valide.

`ConsoleOptionParser::addOptions`(*array $options*)

Si vous avez un tableau avec plusieurs options, vous pouvez utiliser `$parser->addOptions()` pour ajouter plusieurs options en une fois. :

```
$parser->addOptions(array(
    'node', array('short' => 'n', 'help' => 'The node to create'),
    'parent' => array('short' => 'p', 'help' => 'The parent node')
));
```

Comme avec toutes les méthodes de construcion de ConsoleOptionParser, addOptions peut être utilisée comme une partie de la chaîne de méthode courante.

Validation des options

Les options peuvent être fournies avec un ensemble de choix un peu comme les arguments de position peuvent l'être. Quand une option a défini les choix, ceux-ci sont les seuls choix valides pour une option. Toutes les autres valeurs vont lancer une `InvalidArgumentException` :

```
$parser->addOption('accept', array(
    'help' => 'What version to accept.',
    'choices' => array('working', 'theirs', 'mine')
));
```

Utiliser les options boléennes

Les options peuvent être définies en options boléennes, qui sont utiles quand vous avez besoin de créer des options de flag. Comme les options par défaut, les options boléennes les incluent toujours dans les paramètres parsés. Quand les flags sont présents, ils sont définis à true, quand ils sont absents à false :

```
$parser->addOption('verbose', array(
    'help' => 'Enable verbose output.',
    'boolean' => true
));
```

L'option suivante fera que `$this->params['verbose']` sera toujours disponible. Cela vous permet d'oublier `empty()` ou `isset()` vérifie pour les flags de boléens :

```
if ($this->params['verbose']) {
    // faire quelque chose
}
```

Puisque les options boléennes sont toujours définies à `true` ou `false`, vous pouvez omettre les méthodes de vérification supplémentaires.

Ajouter des sous-commandes

`ConsoleOptionParser::addSubcommand`(*$name*, *$options = array()*)

Les applications de Console sont souvent faites de sous-commandes, et ces sous-commandes nécessiteront un parsing d'options spéciales et ont leur propre aide. Un exemple parfait de cela est `bake`. Bake est fait de plusieurs tâches séparées qui ont toutes leur propre aide et options. `ConsoleOptionParser` vous permet

de définir les sous-commandes et de fournir les parsers d'option spécifiques donc le shell sait comment parser les commandes pour ses tâches :

```
$parser->addSubcommand('model', array(
    'help' => 'Bake a model',
    'parser' => $this->Model->getOptionParser()
));
```

Ce qui est au-dessus est un exemple de la façon dont vous pouvez fournir de l'aide et un parser d'option spécialisé pour une tâche du shell. En appelant le `getOptionParser()` de la tâche, nous n'avons pas à dupliquer la génération du parser d'option, ou mixer les concernés dans notre shell. Ajoutez des sous-commandes dans cette façon a deux avantages. Premièrement, cela laisse votre shell documenter facilement ces sous-commandes dans l'aide générée, et cela vous permet aussi un accès facile à l'aide de la sous-commande. Avec la sous-commande créée ci-dessus, vous pouvez appeler `cake myshell --help` et regarder la liste des sous-commandes, et aussi lancer `cake myshell model --help` pour voir l'aide uniquement pour la tâche model.

Quand vous définissez une sous-commande, vous pouvez utiliser les options suivantes :
- `help` - Le texte d'aide pour la sous-commande.
- `parser` - Un ConsoleOptionParser pour la sous-commande. Cela vous permet de créer des méthodes de parsers d'options spécifiques. Quand l'aide est générée pour une sous-commande, si un parser est présent, il sera utilisé. Vous pouvez aussi fournir le parser en tableau qui est compatible avec `ConsoleOptionParser::buildFromArray()`

Ajouter des sous-commandes peut être fait comme une partie de la chaîne de méthode courante.

Construire un ConsoleOptionParser à partir d'un tableau

`ConsoleOptionParser::`**`buildFromArray`**(*$spec*)

Comme mentionné précédemment, lors de la création de parsers d'option de la sous-commande, vous pouvez définir le parser spec en tableau pour cette méthode. Ceci peut faciliter la construction de parsers de sous-commande, puisque tout est un tableau :

```
$parser->addSubcommand('check', array(
    'help' => __('Check the permissions between an ACO and ARO.'),
    'parser' => array(
        'description' => array(
            __("Use this command to grant ACL permissions. Once executed, the ARO "),
            __("specified (and its children, if any) will have ALLOW access to the"),
            __("specified ACO action (and the ACO's children, if any).")
        ),
        'arguments' => array(
            'aro' => array('help' => __('ARO to check.'), 'required' => true),
            'aco' => array('help' => __('ACO to check.'), 'required' => true),
            'action' => array('help' => __('Action to check'))
        )
    )
));
```

A l'intérieur du parser spec, vous pouvez définir les clés pour `definition`, `arguments`, `options`, et `epilog`. Vous ne pouvez pas définir les sous-commandes dans un constructeur

de type tableau. Les valeurs pour les arguments, et les options, doivent suivre le format que `ConsoleOptionParser::addArguments()` et `ConsoleOptionParser::addOptions()` utilisent. Vous pouvez aussi utiliser buildFromArray lui-même, pour construire un parser d'option :

```
public function getOptionParser() {
    return ConsoleOptionParser::buildFromArray(array(
        'description' => array(
            __("Use this command to grant ACL permissions. Once executed, the ARO "),
            __("specified (and its children, if any) will have ALLOW access to the"),
            __("specified ACO action (and the ACO's children, if any).")
        ),
        'arguments' => array(
            'aro' => array('help' => __('ARO to check.'), 'required' => true),
            'aco' => array('help' => __('ACO to check.'), 'required' => true),
            'action' => array('help' => __('Action to check'))
        )
    ));
}
```

Obtenir de l'aide dans les shells

Avec l'ajout de ConsoleOptionParser, récupérer l'aide des shells est faite d'une façon cohérente et uniforme. En utilisant l'option `--help` ou `-h`, vous pouvez voir l'aide pour tout shell du coeur, et tout shell qui implémente un ConsoleOptionParser :

```
cake bake --help
cake bake -h
```

Les deux généreraient l'aide pour. Si le shell supporte les sous-commandes, vous pouvez obtenir l'aide pour ceux-là d'un façon similaire :

```
cake bake model --help
cake bake model -h
```

Cela vous ramènerait l'aide spécifique pour la tâche model de bake.

Obtenir de l'aide en XML

Lorsque vous réalisez des outils d'automatisation ou de développement qui ont besoin d'interagir avec les shells de CakePHP, il est appréciable d'obtenir de l'aide dans un format parsable par une machine. ConsoleOptionParser peut fournir de l'aide au format XML en définissant un argument supplémentaire :

```
cake bake --help xml
cake bake -h xml
```

Les commandes ci-dessus vont retourner un document XML contenant de l'aide à propos des options, arguments et sous-commandes du shell selectionné. Voici un exemple de documentation :

```
<?xml version="1.0"?>
<shell>
    <command>bake fixture</command>
    <description>Generate fixtures for use with the test suite. You can use
        `bake fixture all` to bake all fixtures.</description>
    <epilog>Omitting all arguments and options will enter into an interactive mode.</epilog
    <subcommands/>
    <options>
        <option name="--help" short="-h" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--verbose" short="-v" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--quiet" short="-q" boolean="1">
            <default/>
            <choices/>
        </option>
        <option name="--count" short="-n" boolean="">
            <default>10</default>
            <choices/>
        </option>
        <option name="--connection" short="-c" boolean="">
            <default>default</default>
            <choices/>
        </option>
        <option name="--plugin" short="-p" boolean="">
            <default/>
            <choices/>
        </option>
        <option name="--records" short="-r" boolean="1">
            <default/>
            <choices/>
        </option>
    </options>
    <arguments>
        <argument name="name" help="Name of the fixture to bake.
            Can use Plugin.name to bake plugin fixtures." required="">
            <choices/>
        </argument>
    </arguments>
</shell>
```

Routing dans shells / CLI

Dans l'interface en ligne de commande (CLI), spécialement vos shells et tasks, `env('HTTP_HOST')` et les autres variables d'environnement spécifique à votre navigateur ne sont pas définis.

Si vous générez des rapports ou envoyez des emails qui utilisent `Router::url()`, ceux-ci vont con-

tenir l'hôte par défaut `http://localhost/` et cela va entrainer des urls invalides. Dans ce cas, vous devrez spécifier le domaine manuellement. Vous pouvez faire cela en utilisant la valeur de Configure `App.fullBaseURL` de votre bootstrap ou config, par exemple.

Pour envoyer des emails, vous devrez fournir à la classe CakeEmail l'hôte avec lequel vous souhaitez envoyer l'email en faisant :

$Email = new CakeEmail() ; $Email->domain('www.example.org') ;

Cela suppose que les ID du message généré sont valides et correspondent au domaine duquel les emails sont envoyés.

API de Shell

class `AppShell`

AppShell peut être utilisée comme une classe de base pour tous vos shells. Elle devrait étendre `Shell`, et être localisée dans `Console/Command/AppShell.php`.

class `Shell` (*$stdout = null*, *$stderr = null*, *$stdin = null*)

Shell est une classe de base pour tous les shells, et fournit un certain nombre de fonctions pour l'intéraction avec l'entrée de l'utilisateur, sortant un texte d'erreurs générées.

property `Shell::$tasks`

Un tableau de tâches que vous voulez charger pour ce shell/task.

property `Shell::$uses`

Un tableau de models qui devrait être chargé pour ce shell/task.

`Shell::clear()`

Efface la sortie courante étant affichée.

`Shell::createFile` (*$path*, *$contents*)

Paramètres

- **$path** (*string*) – Le chemin absolu du fichier que vous voulez créer.
- **$contents** (*string*) – Contenus à mettre dans le fichier.

Crée un fichier dans un chemin donné. Si le Shell est intéractif, un avertissement sera généré, et il sera demandé à l'utilisateur si ils veulent écraser le fichier si il existe déjà. Si la propriété interactive du shell est à false, aucune question ne sera demandée et le fichier sera simplement écrasé.

`Shell::dispatchShell()`

Dispatche une commande vers un autre Shell. Similaire à `Controller::requestAction()` mais pour lancer les shells à partir d'autres shells.

Regardez *Invoquer d'autres shells à partir de votre shell*.

`Shell::err` (*$message = null*, *$newlines = 1*)

Paramètres

- **$method** (*string*) – Le message à afficher.
- **$newlines** (*integer*) – Le nombre de nouvelles lignes qui suivent le message.

Sort une méthode vers `stderr`, fonctionne de la même manière que `Shell::out()`

`Shell::error` (*$title*, *$message = null*)

Paramètres
- **$title** (*string*) – Titre d'une erreur
- **$message** (*string*) – Un message d'erreur en option

Affiche un message d'erreurs formaté et sort de l'application avec le code de statut à 1.

`Shell::getOptionParser()`
Devrait retourner un objet `ConsoleOptionParser`, avec tout sous-parsers pour le shell.

`Shell::hasMethod($name)`
Vérifie si ce shell a une méthode appelable par le nom donné.

`Shell::hasTask($task)`
Vérifie si le shell a une tâche avec le nom fourni.

`Shell::hr($newlines = 0, $width = 63)`

Paramètres
- **$newlines** (*int*) – Le nombre de nouvelles lignes à mettre avant et à la suite de la ligne.
- **$width** (*int*) – La largeur de la ligne à dessiner.

Crée une ligne horizontale précédée et suivie par un nombre de nouvelles lignes.

`Shell::in($prompt, $options = null, $default = null)`

Paramètres
- **$prompt** (*string*) – Le prompt à afficher à l'utilisateur.
- **$options** (*array*) – Un tableau de choix valides que l'utilisateur peut choisir. Choisir une option non valide va forcer l'utilisateur à re-choisir.
- **$default** (*string*) – L'option par défaut si il y en a une.

Cette méthode vous aide à intéragir avec l'utilisateur, et crée des shells intéractifs. Elle va retourner la réponse des utilisateurs au prompt, et vous permet de fournir une liste d'options valides dans laquelle l'utilisateur peut choisir :

```
$selection = $this->in('Red or Green?', array('R', 'G'), 'R');
```

La validation de la sélection est sensible à la casse.

`Shell::initialize()`
Initialize le constructeur du shell pour les sous-classes, permet la configuration de tâches avant l'exécution du shell.

`Shell::loadTasks()`
Charge les tâches défines dans public `Shell::$tasks`.

`Shell::nl($multiplier = 1)`

:param int $multiplier Nombre de fois que la séquence de ligne doit être répétée.

Sort un certain nombre de séquences de nouvelles lignes.

`Shell::out($message = null, $newlines = 1, $level = Shell : :NORMAL)`

Paramètres
- **$method** (*string*) – Le message à afficher.
- **$newlines** (*integer*) – Le nombre de nouvelles lignes qui suivent le message.

– **$level** (*integer*) – Le *Niveaux de sortie de la Console* le plus haut que ce message doit afficher.

La méthode primaire pour la génération de la sortie de l'utilisateur. En utilisant les niveaux, vous pouvez utiliser la façon dont un shell est verbose. out() vous permet aussi d'utiliser les tags de formatage de couleur, ce qui permettra la sortie colorée sur les systèmes qui le supportent. Il y a plusieurs styles intégrées pour la coloration de texte, et vous pouvez définir les votres.
–`error` Messages d'Erreur.
–`warning` Messages d'avertissement.
–`info` Messages d'information.
–`comment` Texte supplémentaire.
–`question` Texte Magenta utilisé pour les prompts de l'utilisateur.
En formattant les messages avec des balises de style, vous pouvez afficher une sortie stylisée :

```
$this->out('<warning>This will remove data from the filesystems.</warning>');
```

Par défaut sur les systèmes *nix, les objets ConsoleOutput ont par défaut une sortie colorée. Sur les systèmes windows, la sortie brute est las sortie par défaut sauf si la variable d'environnement `ANSICON` est présente.

`Shell::`**`runCommand`**(*$command*, *$argv*)

Lance le Shell avec argv fourni.

Délégue les appels aux tâches et résoud les méthodes dans la classe. Les commandes sont regardées avec l'ordre suivant :
–Méthode sur le shell.
–Correspondance du nom de la tâche.
–méthode main().
Si un shell intégre une méthode main(), toute appel de méthode perdu sera envoyyé à main() avec le nom de méthode original dans argv.

`Shell::`**`shortPath`**(*$file*)

Rend les chemins de fichier absolus plus faciles à lire.

`Shell::`**`startup`**()

Démarre le Shell et affiche le message d'accueil. Permet de vérifier et configurer avant de faire la commande ou l'exécution principale.

Redéfinit cette méthode si vous voulez retirer l'information de bienvenue, ou sinon modifier le pre-command flow.

`Shell::`**`wrapText`**(*$text*, *$options = array()*)

Entoure un block de texte. Vous permet de configurer la largeur, et d'indenter un block de texte.

Paramètres

– **$text** (*string*) – The text to format
– **$options** (*array*) –
  – `width` La largeur à entourer. Par défaut à 72.
  – `wordWrap` Entoure seulement les espaces de mots. Par défaut à true.
  – `indent` Indente le texte avec la chaîne de caractère fournie. Par défaut à null.

Plus de sujets

Éxecuter des Shells en tâche cron (cronjob)

Une chose habituelle à faire avec un shell, c'est de l'exécuter par une tâche cron pour nettoyer la base de données une fois de temps en temps ou pour envoyer des newsletters.

> # m h dom mon dow command */5 * * * * cd /full/path/to/app && Console/cake myshell myparam

Génération de code avec Bake

La console Bake de CakePHP est un autre outil permettant de réaliser son application rapidement. La console Bake peut créer chacun des ingrédients basiques de CakePHP : models, vues et controllers. Et nous ne parlons pas seulement des squelettes de classes : Bake peut créer une application fonctionnelle complète en seulement quelques minutes. En réalité, Bake est une étape naturelle à suivre une fois qu'une application a été prototypée.

Ceux qui sont novices avec Bake (spécialement les utilisateurs de Windows) pourront trouver le Bake screencast (http ://tv.cakephp.org/video/gwoo/2010/12/24/setting_up_the_cakephp_console_on_windows) utile pour paramétrer les choses avant de continuer.

Suivant la configuration de votre installation, vous devrez peut être donner les droits d'exécution au script bash cake ou l'appeler avec la commande ./cake bake. La console cake est exécutée en utilisant le CLI PHP (Interface de Ligne de Commande). Si vous avez des problèmes en exécutant ce script, vérifiez que le CLI PHP est installé et qu'il a les modules adéquats autorisés (ex : MySQL). Certains utilisateurs peuvent aussi rencontrer des problèmes si la base de donnée host est 'localhost' et devront essayer '127.0.0.1' à la place. Cela peut causer des soucis avec le CLI PHP.

En exécutant Bake la première fois, vous serez invité à créer un fichier de configuration de la base de données, si vous n'en avez pas créées auparavant.

Après que vous avez créé un fichier de configuration de base de données, exécuter Bake vous présentera les options suivantes

```
---------------------------------------------------------------
App : app
Path: /path-to/project/app (Chemin: /chemin/vers/app/du/projet)
---------------------------------------------------------------
Interactive Bake Shell
---------------------------------------------------------------
[D]atabase Configuration
[M]odel
[V]iew
[C]ontroller
[P]roject
[F]ixture
[T]est case
[Q]uit
What would you like to Bake? (D/M/V/C/P/F/T/Q)
>
```

Sinon, vous pouvez exécuter chacune de ces commandes directement depuis la ligne de commande

```
$ cake bake db_config
$ cake bake model
$ cake bake view
$ cake bake controller
$ cake bake project
$ cake bake fixture
$ cake bake test
$ cake bake plugin plugin_name
$ cake bake all
```

Modifier le HTML par défaut produit par les templates de bake

Si vous souhaitez modifier la sortie HTML par défaut produite par la commande "bake", suivez les étapes simples suivantes :

Pour fabriquer des vues sur mesure

1. Aller dans le dossier : lib/Cake/Console/Templates/default/views.
2. Remarquez les 4 fichiers ici.
3. Copier les dans le dossier : app/Console/Templates/[themename]/views.
4. Faire les changements pour la sortie HTML pour contrôler la façon dont "bake" fabrique vos vues.

La partie du chemin `[themename]` est le nom du theme de bake que vous créez. Les noms des thèmes de Bake doivent être uniques, donc n'utilisez pas 'default'.

Pour baker des projets personnalisés

Allez dans : lib/Cake/Console/Templates/skel Trouvez-y la base des fichiers d'application. Copiez les dans votre : app/Console/Templates/skel Faîtes des changements vers la sortie HTML pour contrôler la façon dont "bake" va construire vos vues. Passez le paramètre de squelette du chemin à la tâche du projet.

```
cake bake project -skel Console/Templates/skel
```

Note :
- Vous devez lancer la tâche du projet spécifique `cake bake project` afin que le paramètre du chemin soit passé.
- Le chemin du template est relatif au chemin courant de l'Interface de Commande en Ligne.
- Puisque le chemin complet du squelette doit être entré manuellement, vous pouvez spécifier n'importe quel dossier avec le template que vous souhaitez construire, ainsi que l'utilisation de plusieurs templates. (Sauf si Cake commence par outrepasser le dossier squelette comme il fait pour les vues)

Amélioration de Bake dans la version 1.3

Dans CakePHP 1.3, bake a connu une révision importante, avec le rajout de fonctionnalités et améliorations.
- Deux nouvelles tâches (FixtureTask and TestTask) sont accessibles à partir du menu principal de bake.
- Une troisième tâche (TemplateTask) a été rajoutée pour l'utilisation dans vos shells.
- Toutes ces différentes tâches de bake vous permettent maintenant d'utiliser d'autres connections de bake que le 'default'. Utilisez le paramètre `-connection`.
- Le support de Plugin a été fortement amélioré. Vous pouvez maintenant utiliser `-plugin NomDuPlugin` ou `Plugin.class`.
- Les Questions ont été clarifiées, et sont plus facilement compréhensibles.
- Les validations multiples sur les models ont été ajoutées.
- Les associations des models sur eux-mêmes utilisant `parent_id` sont maintenant détectées. Par exemple, si votre model est appelé Thread, une association ParentThread et ChildThread sera créée.
- Fixtures et Tests peuvent être 'cuits' séparément.
- Les Tests 'Cuits' incluent autant de fixtures connues, ainsi que la détection des plugins (La détection plugin ne fonctionne pas avec PHP4).

Ainsi, avec cette liste de fonctionnalités, nous allons prendre le temps de regarder certaines nouvelles commandes, certains nouveaux paramètres et les fonctionnalités mises à jour.

Nouveaux FixtureTask, TestTask et TemplateTask.

Fixture et le test baking étaient un peu ardus dans le passé. Vous pouviez seulement générer des tests quand vous bakiez des classes, et les fixtures pouvaient seulement être générées quand on bakait les models. Cela faisait que l'ajout ultérieur de tests à vos applications ou même la régénération de fixtures avec de nouveaux schémas étaient difficiles. Dans 1.3, nous avons séparé Fixture et la fabrication des Tests en tâches différentes. Cela vous permet de les relancer et de regénérer des tests et fixtures à n'importe quel moment dans votre processus de développement.

En plus d'être reconstructibles à n'importe quel moment, les tests cuits sont maintenant capable de trouver autant de fixtures que possible. Dans le passé, tester impliquait souvent de se battre à travers de nombreuses erreurs 'Manque la table'. Avec une detection des fixtures plus poussée, nous espérons rendre le test plus simple plus accessible.

Les cas de test générent aussi des méthodes squelettes de test pour chaque méthode publique non héritée dans vos classes. Vous enlevant une étape supplémentaire.

`TemplateTask` est une tâche en arrière plan, et elle gère la génération des fichiers à partir de templates. Dans les versions précédentes de CakePHP les vues cuites étaient basées sur des templates, mais tout le reste du code ne l'était pas. Avec 1.3, presque tout le contenu dans les fichiers générés par bake sont contrôlés par les templates et la `TemplateTask`.

`FixtureTask` ne génére plus seulement les fixtures avec les données factices mais en utilisant les options interactives ou l'option `-records` vous pouvez activer la génération de fixture en utilisant les données live.

Nouvelle commande bake De nouvelles commandes ont été ajoutées pour rendre le baking plus facile et plus rapide. Les bakings des controllers, Models et Vues ont tous la fonctionnalité de sous-commande `all`, qui construit tout en une fois et reconstruit rapidement et facilement.

```
cake bake model all
```

Bakerait tous les models pour une application en une fois. De même, `cake bake controller all` bakerait tous les controllers et `cake bake view all` générerait tous les fichiers vues. Les paramètres de

la tâche `ControllerTask` ont aussi changé. `cake bake controller scaffold` est maintenant `cake bake controller public`. `ViewTask` a eu un drapeau `-admin` ajouté, en utilisant `-admin` cela vous autorise à baker les vues pour les actions qui commencent par `Routing.admin`.

Comme mentionné avant `cake bake fixture` et `cake bake test` sont nouveaux, et ont plusieurs sous-commandes chacun. `cake bake fixture all` va regénérer tous les fixtures basiques pour votre application. Le paramètre `-count` vous autorise à configurer le nombre d'enregistrements faux qui sont créés. En lançant la tâche de fixture de façon interactive, vous pouvez générer les fixtures en utilisant les données dons vos tables live. Vous pouvez utiliser `cake bake test <type> <class>` pour créer les cas de test pour les objets déjà crées dans votre app. Le type doit être l'un des types standards de CakePHP ('component', 'controller', 'model', 'helper', 'behavior') mais peut ne pas exister. Les classes doivent être un objet existant d'un type choisi.

Des templates en abondance

Une nouveauté dans bake pour 1.3 est l'addition de plus de templates. Dans 1.2, les vues bakées utilisaient les templates qui pouvaient être changés pour modifier les fichiers vues bakés générées. Dans 1.3, les templates sont utilisés pour générer toute sortie de bake générée. Il y a des templates séparés poour les controllers, les ensembles d'action des controllers, les fixtures, les models, les cas de test, et les fichiers de vue de 1.2. Comme de plus en plus de templates, vous pouvez aussi avoir des ensembles de template multiple ou, de thèmes bakés. Les thèmes bakés peuvent être fournis dans votre app, ou dans une partie des plugins. Un exemple de chemin de plugin pouf le thème baké serait `app/Plugin/BakeTheme/Console/Templates/dark_red/`. Un thème d'app bakée appelé `blue_bunny` serait placé dans `app/Console/Templates/blue_bunny`. Vous pouvez regarder dans `lib/Cake/Console/Templates/default/` pour voir quels répertoires et fichiers sont requis d'un thème baké. Cependant, comme les fichiers vues, si votre thème baké n'implémente pas un template, les autres thèmes installés seront vérifiés jusqu'à ce que le template correct soit trouvé.

Support de plugins additionels.

Nouveau dans 1.3 sont les chemins additionnels pour spécifier les noms de plugin quand on utilise bake. En plus de `cake bake plugin Todo controller Posts`, il y a deux nouvelles formes. `cake bake controller Todo.Posts` et `cake bake controller Posts -plugin Todo`. Le paramètre de plugin peut aussi exister en utilisant le bake interactif. `cake bake controller -plugin Todo`, par exemple vous autorisera à utiliser le bake interactif pour ajouter des controllers à votre plugin Todo. Des chemins de plugin supplémentaires / multiples sont aussi supportés. Dans le passé, bake nécessitait que le plugin soit dans app/plugins. Dans 1.3, bake trouvera le chemin du plugin pour le plugin nommé, et y ajoutera les fichiers.

Gestion des Schémas et migrations

Le shell Schema fournit une fonctionnalité pour créer des objets, des dumps sql et pour créer et restaurer des vues instantanées de votre base de données.

Générer et utiliser les fichiers Schema

Un fichier de génération de schéma vous permet de facilement transporter un schéma de base de données agnostique. Vous pouvez générer un fichier de schéma de votre base de données en utilisant :

```
$ Console/cake schema generate
```

Cela générera un fichier schema.php dans votre dossier `app/Config/Schema`.

Note : Le shell schema n'utilise que les tables pour lesquelles des models sont définis. Pour forcer le shell à considérer toutes les tables, vous devez ajouter l'option `-f` à votre ligne de commande.

Pour reconstruire plus tard votre schéma de base de données à partir d'un fichier schema.php précédemment réalisé, lancez

```
$ Console/cake schema create
```

Cela va supprimer et créer les tables en se basant sur le contenu de schema.php.

Les fichiers de schéma peuvent aussi être utilisés pour générer des dumps sql. Pour générer un fichier sql comprennant les définitions `CREATE TABLE`, lancez :

```
$ Console/cake schema dump --write filename.sql
```

nomdufichier.sql est le nom souhaité pour le fichier contenant le dump sql. Si vous omettez nomdufichier.sql, le dump sql sera affiché sur la console, mais ne sera pas écrit dans un fichier.

callbacks de CakeSchema

Après avoir généré un schema, vous voudrez peut-être insérer des données de départ à des tables de votre application. Ceci peut être accompli avec les callbacks de CakeSchema. Chaque fichier de schema est généré avec les méthodes `before($event = array())` et `after($event = array())`.

Le paramètre `$event` contient un tableau avec deux clés. Une pour dire si une table a été supprimée ou créée et une autre pour les erreurs. Exemples :

```
array('drop' => 'posts', 'errors' => null)
array('create' => 'posts', 'errors' => null)
```

Ajouter des données à une table posts par exemple donnerait ceci :

```
App::uses('Post', 'Model');
public function after($event = array()) {
    if (isset($event['create'])) {
        switch ($event['create']) {
            case 'posts':
                App::uses('ClassRegistry', 'Utility');
                $post = ClassRegistry::init('Post');
                $post->create();
                $post->save(
                    array('Post' =>
                        array('title' => 'CakePHP Schema Files')
                    )
                );
```

```
                break;
            }
        }
}
```

Les callbacks `before()` et `after()` se lancent à chaque fois qu'une table est créée ou supprimée sur le schema courant.

Quand vous insérez des données à plus d'une table, vous devrez faire un flush du cache de la base de données après la création de chaque table. Le Cache peut être désactivé en configurant `$db->cacheSources = false` dans l'action before().

```
public $connection = 'default';

public function before($event = array()) {
    $db = ConnectionManager::getDataSource($this->connection);
    $db->cacheSources = false;
    return true;
}
```

Migrations avec le shell schema de CakePHP

Les migrations permettent de "versionner" votre schéma de base de données, de telle façon que lorsque vous développez des fonctionnalités, vous avez une méthode facile et élégante pour relever les modifications apportées à votre base. Les migrations sont réalisées soit grâce aux fichiers de schémas, soit grâce aux vues instantanées. Versionner un fichier de schéma avec le shell schema est assez facile. Si vous avez déjà un fichier schema créé en utilisant

```
$ Console/cake schema generate
```

Vous aurez alors les choix suivants

```
Generating Schema... (Génération du Schema)
Schema file exists. (Le fichier schema existe)
 [O]verwrite (Ecraser)
 [S]napshot  (Vue instantanée)
 [Q]uit      (Quitter)
Would you like to do? (o/s/q) (Que voulez vous faire?)
```

Choisir [s] (snapshot - vue instantanée) va créer un fichier schema.php incrémenté. Ainsi, si vous avez schema.php, cela va créer schema_2.php et ainsi de suite. Vous pouvez ensuite restaurer chacun de ces schémas en utilisant

```
$ cake schema update -s 2
```

Où 2 est le numéro de la vue instantanée que vous voulez exécuter. Le shell vous demandera de confirmer votre intention d'exécuter les définitions `ALTER` qui représentent les différences entre la base existante et le fichier de schéma exécuté à ce moment.

Vous pouvez effectuer un lancement d'essai ("dry run") en ajoutant `--dry` à votre commande.

Exemple d'application

Créer un schéma et committer

Sur un projet qui utilise le versioning, l'utilisation du schema cake suivrait les étapes suivantes :

1. Créer ou modifier les tables de votre base de données.
2. Exécuter cake schema pour exporter une description complète de votre base de données.
3. Committer et créer ou modifier le fichier schema.php :

```
$ # Une fois que votre base de données a été mise à jour
$ Console/cake schema generate
$ git commit -a
```

Note : Si le projet n'est pas versionné, la gestion des schémas se fera à travers des vues instantanées. (voir la section précédente pour gérer les vues instantanées)

Récupérer les derniers changements

Quand vous récupérez les derniers changements de votre répertoire, et découvrez des changements dans la structure de la base de données (par exemple vous avez un message d'erreur disant qu'il manque une table) :

1. Exécuter cake schema pour mettre à jour votre base de données :

```
$ git pull
$ Console/cake schema create
$ Console/cake schema update
```

Toutes ces opérations peuvent être faîtes en mode sans écriture.

Revenir en arrière

Si à un moment donné vous avez besoin de revenir en arrière et de retourner à un état précédent à votre dernière mise à jour, vous devez être informé que ce n'est pas pour l'instant pas possible avec cake schema.

Plus précisemment, vous ne pouvez pas supprimer automatiquement vos tables une fois qu'elles ont été créees.

L'utilisation de `update` supprimera, au contraire, n'importe quel champ qui différera de votre fichier schema :

```
$ git revert HEAD
$ Console/cake schema update
```

Ceci vous proposera les choix suivants :

```
The following statements will run. (Les requêtes suivantes vont être exécutées)
ALTER TABLE `roles`
DROP `position`;
Are you sure you want to alter the tables? (y/n) (Êtes vous sur de vouloir modifier les ta
[n] >
```

Shell I18N

La fonctionnalité i18n de CakePHP utilise les fichiers po (http ://en.wikipedia.org/wiki/GNU_gettext) comme source de traduction. Cela les rend faciles à intégrer avec des outils tels que poedit (http ://www.poedit.net/) ou d'autres outils habituels de traduction.

Le Shell de i18n fournit une façon rapide et simple de générer les fichiers template po. Les fichiers templates peuvent être donnés aux traducteurs afin qu'ils traduisent les chaînes de caractères dans votre application. Une fois que votre traduction est faîte, les fichiers pot peuvent être fusionnés avec les traductions existantes pour aider la mise à jour de vos traductions.

Générer les fichiers POT

Les fichiers POT peuvent être généres pour une application existante en utilisant la commande `extract`. Cette commande va scaner toutes les fonctions de type `__()` de l'ensemble de votre application, et extraire les chaînes de caractères. Chaque chaîne unique dans votre application sera combinée en un seul fichier POT :

```
./Console/cake i18n extract
```

La commande du dessus va lancer le shell d'extraction. En plus de l'extraction des chaînes de caractères des méthodes `__()`, Les messages de validation des modèles vont aussi être extraits. Le résultat de cette commande va être la création du fichier `app/Locale/default.pot`. Vous utilisez le fichier pot comme un template pour créer les fichiers po. Si vous créez manuellement les fichiers po à partir du fichier pot, pensez à bien corriger le `Plural-Forms` de la ligne d'en-tête.

Générer les fichiers POT pour les plugins

Vous pouvez générer un fichier POT pour un plugin spécifique en faisant :

```
./Console/cake i18n extract --plugin <Plugin>
```

Cela générera les fichiers POT requis utilisés dans les plugins.

Messages de Validation de Model

Vous pouvez définir le domaine à utiliser pour les messages de validation extraits dans vos models. Si le model a toujours une propriété `$validationDomain`, le domaine de validation donnée va être ignoré :

```
./Console/cake i18n extract --validation-domain validation_errors
```

Vous pouvez aussi éviter que le shell n'extrait des messages de validation :

```
./Console/cake i18n extract --ignore-model-validation
```

Exclure les fichiers

Vous pouvez passer une liste de dossiers séparée par une virgule que vous souhaitez exclure. Tout chemin contenant une partie de chemin avec les valeurs fournies sera ignoré :

```
./Console/cake i18n extract --exclude Test,Vendor
```

Eviter l'écrasement des avertissements pour les fichiers POT existants

Introduit dans la version 2.2. En ajoutant –overwrite, le script de shell ne va plus vous avertir si un fichier POT existe déjà et va écraser par défaut :

```
./Console/cake i18n extract --overwrite
```

Extraire les messages des librairies du coeur de CakePHP

Introduit dans la version 2.2. Par défaut, le script de shell d'extraction va vous demander si vous souhaitez extraire les messages utilisés dans les librairies du coeur de CakePHP. Définissez –extract-core à yes ou no pour définir le comportement par défaut.

```
./Console/cake i18n extract --extract-core yes

ou

./Console/cake i18n extract --extract-core no
```

Créer les tables utilisées par TranslateBehavior

Le shell i18n peut aussi être utilisé pour initialiser les tables par défaut utilisées par `TranslateBehavior` :

```
./Console/cake i18n initdb
```

Cela va créer la table `i18n` utilisée par le behavior Translate.

Shell ACL

Le Shell Acl est utile pour gérer et inspecter les enregistrements de vos bases de données Acl. Il est souvent plus pratique que l'ajout de modifications ponctuelles dans les controllers.

La plupart des sous-commandes shell acl implique le référencement des noeuds aco/aro. Comme il y a deux 'formes' de ces noeuds, il existe deux notation dans le shell :

```
# Un Model + référence de la clé étrangère
./Console/cake acl view aro Model.1

# Un chemin alias de référence
./Console/cake acl view aco root/controllers
```

Utiliser `.` indique que vous allez utiliser une référence d'enregistrement liée au style, tandis qu'utiliser un `/` indique un chemin alias.

Installer les tables de la base de données

Avant d'utiliser la base de données ACL, vous aurez besoin de configurer les tables. Vous pouvez le faire en utilisant :

```
./Console/cake acl initdb
```

Créer et supprimer les noeuds

You pouvez utiliser les sous-commandes de création et de suppression pour créer et supprimer des noeuds :

```
./Console/cake acl create aco controllers Posts
./Console/cake acl create aco Posts index
```

Cette commande crée un enregistrement aco en utilisant un chemin alias. Vous pouvez aussi faire comme ce qui suit :

```
./Console/cake acl create aro Group.1
```

Pour créer un noeud aro pour le Groupe dont l'id est = 1.

Accorder et refuser l'accès

Utilisez la commande d'accès pour accorder les permssions ACL. Une fois executée, l'ARO spécifié (et ses enfants, si il en a) aura un accès AUTORISÉ à l'action ACO spécifié (et les enfants de l'ACO, si il y en a) :

```
./Console/cake acl grant Group.1 controllers/Posts
```

La commande ci-dessus accorde tous les privilèges. Vous pouvez n'accorder que les privilèges de lecture en utilisant la commande suivante :

```
./Console/cake acl grant Group.1 controllers/Posts read
```

Refuser une permission fonctionne exactement de la même façon. La seule différence est le remplacement de 'grant' en 'deny'.

Vérification des permissions

Utilisez les commandes pour accorder les permissions ACL. Une fois exécutées, l'ARO spécifié (et ses enfants, s'il en a) aura un accès AUTORISÉ pour l'action ACO spécifiée (et les enfants de l'ACO, si ils existent) :

```
./Console/cake acl check Group.1 controllers/Posts read
```

La sortie sera soit `success``(succès), soit ``not allowed` (non autorisé).

Voir l'arbre de noeuds

La commande view permet de voir les arbres des ARO et des ACO. Le paramètre optionnel 'node' permet de retourner seulement une portion de l'arbre demandé :

```
./Console/cake acl view
```

Shell Testsuite

Une fois que vous avez commencé à écrire *Tests* vous pouvez les lancer en utilisant le shell testsuite.

Pour plus d'informations sur les utilisations classiques du shell testsuite, allez voir *Lancer les tests à partir d'une ligne de commande*. Modifié dans la version 2.1 : Le shell `test``a été ajouté dans 2.1. Le shell ``testsuite` 2.0 est toujours disponible mais la nouvelle syntaxe est préférée.

Mise à jour shell

La mise à jour shell va faire quasiment tout le boulot pour mettre à jour vos applications cakePHP de la version 1.3 à 2.0.

Pour lancer la mise à jour :

```
./Console/cake upgrade all
```

Si vous voulez voir ce que le shell va faire sans modifier les fichiers, faites d'abord une exécution à blanc avec –dry-run :

```
./Console/cake upgrade all --dry-run
```

Pour mettre à jour vos Plugins, lancer la commande :

```
./Console/cake upgrade all --plugin YourPluginName
```

Il est aussi possible de lancer chaque mise à jour individuellement. Pour voir toutes les étapes possibles, lancer la commande :

```
./Console/cake upgrade --help
```

Ou visiter les docs de l'api pour plus d'informations : http ://api20.cakephp.org/class/upgrade-shell

Mise à jour de votre App

Vous trouverez ici un guide pour vous aider à mettre à jour votre application CakePHP 1.3 vers cakePHP 2.x en utilisant le shell upgrade. La structure de dossiers de votre application 1.3 ressemble à cela :

```
monsiteweb/
    app/              <- Votre App
    cake/             <- Version de CakePHP 1.3
    plugins/
    vendors/
    .htaccess
    index.php
```

La première étape est de télécharger (ou de faire `git clone`) la nouvelle version de CakePHP dans un autre dossier en dehors de votre dossier `monsiteweb`, que nous appellerons `cakephp`. Nous ne souhaitons pas que le dossier téléchargé `app` écrase votre dossier app. Maintenant, il est grand temps de faire une sauvegarde de votre dossier app, par exemple :`cp -R app app-backup`.

Copiez le dossier `cakephp/lib` dans votre dossier `monsiteweb/lib` pour mettre à jour la nouvelle version de CakePHP dans votre app, par exemple : `cp -R ../cakephp/lib .`. Symlinking est aussi une bonne alternative pour copier, par exemple. : `ln -s /var/www/cakephp/lib`.

Avant de lancer notre shell de mise à jour, nous avons aussi besoin des nouveaux scripts de console. Copier le dossier `cakephp/app/Console` dans le dossier `monsiteweb/app`, exemple. : `cp -R ../cakephp/app/Console ./app`.

La structure de votre dossier devrait ressembler à cela maintenant :

```
monsiteweb/
    app/              <- Votre App
        Console/      <- Dossiers Copiés app/Console
    app-backup/       <- Sauvegarde de votre App
    cake/             <- 1.3 Version de CakePHP
    lib/              <- 2.x Version de CakePHP
        Cake/
    plugins/
    vendors/
    .htaccess
    index.php
```

Maintenant nous pouvons lancer la mise à jour shell en tapant `cd` puis le chemin vers votre app et en lancant la commande :

```
./Console/cake upgrade all
```

Cela fera la **plupart** du travail pour mettre à jour votre app vers 2.x. Vérifiez dans votre dossier `app` mis à jour. Si tout a l'air bien, félicitez vous vous-mêmes et supprimez votre dossier `mywebsite/cake`. Bienvenue dans la version 2.x !

CHAPITRE 10

Développement

Dans cette section, nous couvrirons les différents aspects du développement d'une application CakePHP. Les sujets tels que la Configuration, la gestion des erreurs & des exceptions, le débugging et le test seront couverts.

Configuration

Configurer une application CakePHP c'est du gâteau. Après avoir installé CakePHP, la création d'une application web basique nécessite seulement que vous définissiez une configuration à la base de données.

Il y a toutefois d'autres étapes optionnelles de configuration que vous pouvez suivre afin de tirer profit de l'architecture flexible de CakePHP. Vous pouvez facilement ajouter des fonctionnalités provenant du cœur de CakePHP, configurer des mappings URLs supplémentaires/différentes (routes) et définir des inflexions supplémentaires/différentes.

Configuration de la Base de Données

CakePHP s'attend à trouver les détails de configuration de la base de données dans le fichier `app/Config/database.php`. Un exemple de fichier de configuration de base de données peut être trouvé dans `app/Config/database.php.default`. Une configuration basique complète devrait ressembler à quelque chose comme cela :

```
class DATABASE_CONFIG {
    public $default = array(
        'datasource'   => 'Database/Mysql',
        'persistent'   => false,
        'host'         => 'localhost',
        'login'        => 'cakephpuser',
        'password'     => 'c4k3roxx!',
        'database'     => 'my_cakephp_project',
        'prefix'       => ''
    );
}
```

Le tableau de connexion $default est utilisé tant qu'aucune autre connexion n'est spécifiée dans un model, par la propriété `$useDbConfig`. Par exemple, si mon application a une base de données pré-existante, outre celle par défaut, je pourrais l'utiliser dans mes models, en créant un nouveau tableau de connexion à la base de données, intitulé $ancienne, identique au tableau $default, puis en initialisant la propriété `public $useDbConfig = 'ancienne';` dans les models appropriés.

Complétez les couples clé/valeur du tableau de configuration pour répondre au mieux à vos besoins.

datasource Le nom de la source de données pour lequel ce tableau de configuration est destiné. Exemples : Database/Mysql, Database/Sqlserver, Database/Postgres, Database/Sqlite. Vous pouvez utiliser la *plugin syntax* pour indiquer la source de données du plugin à utiliser.

persistent Indique si l'on doit ou non utiliser une connexion persistante à la base.

host Le nom du serveur de base de données (ou son adresse IP).

login Le nom d'utilisateur pour ce compte.

password Le mot de passe pour ce compte.

database Le nom de la base de données à utiliser pour cette connexion.

prefix (*optionel*) La chaîne qui préfixe le nom de chaque table dans la base de données. Si vos tables n'ont pas de préfixe, laissez une chaîne vide pour cette valeur.

port (*optionel*) Le port TCP ou le socket Unix utilisé pour se connecter au serveur.

encoding Indique quel jeu de caractères utiliser pour envoyer les instructions SQL au serveur. Ces valeurs pour l'encodage par défaut de la base de données sont valables pour toutes les bases autres que DB2. Si vous souhaitez utiliser l'encodage UTF-8 avec des connexions mysql/mysqli, vous devez écrire 'utf8' sans le tiret.

schema Utilisé dans les paramètres d'une base PostgreSQL pour indiquer quel schéma utiliser.

datasource Source de données Non-DBO à utiliser, ex : 'ldap', 'twitter'.

unix_socket Utilisé par les pilotes qui le supportent pour connecter via les fichiers socket unix. Si vous utilisez postgres et voulez utiliser les sockets unix, laissez la clé host vide.

ssl_key Le chemin vers le fichier de clé SSL. (Seulement supporté par MySQL, nécessite PHP 5.3.7+).

ssl_cert Le chemin vers le fichier de certificat SSL. (Seulement supporté par MySQL, nécessite PHP 5.3.7+).

ssl_ca Le chemin vers l'autorité de certification SSL. (Seulement supporté par MySQL, nécessite PHP 5.3.7+).

settings Un tableau de clé/valeur qui doit être envoyé à la base de données du serveur en tant que commandes `SET` quand la connection est créée. Cette option est seulement supportée par MySQL, Postgres, et SQLserver en ce moment.

Modifié dans la version 2.4 : Les clés `settings`, `ssl_key`, `ssl_cert` et `ssl_ca` ont été ajoutées dans 2.4.

Note : Le paramétrage du préfixe est valable pour les tables, **pas** pour les models. Par exemple, si vous créez une table de liaison entre vos models Apple et Flavor, vous la nommerez "prefix_apples_flavors" (et **non pas** "prefix_apples_prefix_flavors") et vous paramétrerez votre propriété "prefix" sur 'prefix_'.

A présent, vous aurez peut-être envie de jeter un œil aux *Conventions de CakePHP*. Le nommage correct de vos tables (et de quelques colonnes en plus) peut vous rapporter quelques fonctionnalités supplémentaires et

vous éviter trop de configuration. Par exemple, si vous nommer votre table big_boxes, votre model BigBox, votre controller BigBoxesController, tout marchera ensemble automatiquement. Par convention, utilisez les underscores, les minuscules et les formes plurielles pour les noms de vos tables - par exemple : bakers, pastry_stores, et savory_cakes.

Chemins de Classe Supplémentaires

Il est occasionnellement utile d'être capable de partager des classes MVC entre des applications sur le même système. Si vous souhaitez le même controller dans les deux applications, vous pouvez utiliser le bootstrap.php de CakePHP pour amener ces classes supplémentaires dans la vue.

En utilisant `App::build()` dans bootstrap.php nous pouvons définir des chemins supplémentaires où CakePHP va recherchez les classes :

```
App::build(array(
    'Model'                      => array('/path/to/models', '/next/path/to/models'),
    'Model/Behavior'             => array('/path/to/behaviors', '/next/path/to/behaviors'),
    'Model/Datasource'           => array('/path/to/datasources', '/next/path/to/datasources
    'Model/Datasource/Database'  => array('/path/to/databases', '/next/path/to/database'),
    'Model/Datasource/Session'   => array('/path/to/sessions', '/next/path/to/sessions'),
    'Controller'                 => array('/path/to/controllers', '/next/path/to/controllers
    'Controller/Component'       => array('/path/to/components', '/next/path/to/components'
    'Controller/Component/Auth'  => array('/path/to/auths', '/next/path/to/auths'),
    'Controller/Component/Acl'   => array('/path/to/acls', '/next/path/to/acls'),
    'View'                       => array('/path/to/views', '/next/path/to/views'),
    'View/Helper'                => array('/path/to/helpers', '/next/path/to/helpers'),
    'Console'                    => array('/path/to/consoles', '/next/path/to/consoles'),
    'Console/Command'            => array('/path/to/commands', '/next/path/to/commands'),
    'Console/Command/Task'       => array('/path/to/tasks', '/next/path/to/tasks'),
    'Lib'                        => array('/path/to/libs', '/next/path/to/libs'),
    'Locale'                     => array('/path/to/locales', '/next/path/to/locales'),
    'Vendor'                     => array('/path/to/vendors', '/next/path/to/vendors'),
    'Plugin'                     => array('/path/to/plugins', '/next/path/to/plugins'),
));
```

Note : Tout chemin de configuration supplémentaire doit être fait en haut du bootstrap.php de votre application. Cela va assurer que les chemins sont disponibles pour le reste de votre application.

Configuration du Coeur

Chaque application dans CakePHP contient un fichier de configuration pour déterminer le comportement interne de CakePHP. `app/Config/core.php`. Ce fichier est une collection de définitions de variables et de constantes de la classe Configure qui déterminent comment votre application se comporte. Avant que nous creusions ces variables particulières, vous aurez besoin d'être familier avec la classe de configuration registry `Configure` de CakePHP.

Configuration du Coeur de CakePHP

La classe *Configure* est utilisée pour gérer un ensemble de variables de configuration du coeur de CakePHP. Ces variables peuvent être trouvées dans `app/Config/core.php`. Ci-dessous se trouve une description de chaque variable et comment elle affecte votre application CakePHP.

debug Change la sortie de debug de CakePHP. 0 = mode Production. Pas de sortie. 1 = Montre les erreurs et les avertissements. 2 = Montre les erreurs, avertissements, et le SQL. [le log SQL est seulement montré quand vous ajoutez $this->element('sql_dump') à votre vue ou votre layout.]

Error Configure le getionnaire d'Error handler utilisé pour gérer les erreurs pour votre application. Par défaut `ErrorHandler::handleError()` est utilisé. Cela affichera les erreurs en utilisant `Debugger`, quand debug > 0 et les logs d'erreurs avec `CakeLog` quand debug = 0.

Sous-clés :

- `handler` - callback - Le callback pour gérer les erreurs. Vous pouvez définir cela à n'importe quel callback, en incluant les fonctions anonymes.
- `level` - int - Le niveau d'erreurs pour lequel vous souhaitez faire une capture.
- `trace` - boolean - Inclut les traces de pile d'erreurs dans les fichiers log.

Exception Configure le gestionnaire Exception utilisé pour les exceptions non attrapées. Par défaut, ErrorHandler : :handleException() est utilisée. Elle va afficher une page HTML pour l'exception, et tant que debug > 0, les erreurs du framework comme Missing Controller seront affichées. Quand debug = 0, les erreurs du framework seront forcées en erreurs génériques HTTP. Pour plus d'informations sur la gestion de d'Exception, regardez la section *Exceptions*.

App.baseUrl Si vous ne souhaitez pas ou ne pouvez pas avoir le mod_rewrite (ou un autre module compatible) et ne pouvez pas le lancer sur votre serveur, vous aurez besoin d'utiliser le système de belles URLs construit dans Cake. Dans `/app/Config/core.php`, décommentez la ligne qui ressemble à cela :

```
Configure::write('App.baseUrl', env('SCRIPT_NAME'));
```

Retirez aussi ces fichiers .htaccess :

```
/.htaccess
/app/.htaccess
/app/webroot/.htaccess
```

Cela fera apparaitre vos URLs de la façon suivante www.example.com/index.php/controllername/actionname/param plutôt que www.example.com/controllername/actionname/param.

Si vous installez CakePHP sur un serveur web autre que Apache, vous pouvez trouver des instructions pour faire fonctionner l'URL rewriting pour d'autres serveurs dans la section *URL Rewriting*.

App.encoding Définit quel encodage votre application utilise. Cet encodage est utilisé pour générer le charset dans le layout, et les entités d'encodage. Il doit correspondre aux valeurs encodées spécifiées pour votre base de données.

Routing.prefixes Décommentez cette définition si vous souhaitez tirer profit des routes préfixées de CakePHP comme admin. Définissez cette variable avec un tableau de noms préfixés de routes que vous voulez utiliser. En savoir plus sur cela plus tard.

Cache.disable Quand défini à true, la mise en cache persistente est désativée côté-site. Cela mettra toutes les lectures/écritures du `Cache` en échec.

Cache.check Si défini à true, active la mise en cache de la vue. L'activation est toujours necéssaire dans les controllers, mais cette variable permet la détection de ces configurations.

Session Contient un tableau de configurations à utiliser pour la configuration de session. La clé par défaut est utilisée pour définir un preset par défaut pour utiliser les sessions, toute configuration déclarée ici va écraser les configurations de la config par défaut.

Sous-clés

- `name` - Le nom du cookie à utiliser. Par défaut 'CAKEPHP'.
- `timeout` - Le nombre de minutes de vie des sessions. Le timeout est géré par CakePHP.
- `cookieTimeout` - Le nombre de minutes de vie des cookies de session.
- `checkAgent` - Voulez-vous que l'user agent soit vérifié quand on démarre les sessions ? Vous voudrez peut-être définir la valeur à false, quand il s'agit de vieilles versions de IE, Chrome Frame ou certains navigateurs et AJAX.
- `defaults` - La configuration par défaut définie à utiliser comme base pour votre session. Il y en a quatre intégrées : php, cake, cache, database.
- `handler` - Peut être utilisé pour activer un gestionnaire de session personnalisé. Attend un tableau de callables, qui peut être utilisé avec *session_save_handler*. L'utilisation de cette option va automatiquement ajouter *session.save_handler* au tableau ini.
- `autoRegenerate` - Activer cette configuration allume un renouveau automatique des sessions, et des ids de session qui changent fréquemment. Regardez `CakeSession::$requestCountdown`.
- `ini` - Un tableau associatif de valeurs ini supplémentaires à définir.

Les paramètres par défaut intégrés sont :

- 'php' - Utilise les configurations définies dans votre php.ini.
- 'cake' - Sauvegarde les fichiers de session dans le répertoire /tmp de CakePHP's /tmp.
- 'database' - Utilise les sessions de base de données de CakePHP.
- 'cache' - Utilise la classe de Cache pour sauvegarder les sessions.

Pour définir un gestionnaire de session personnalisé, sauvegardez le dans `app/Model/Datasource/Session/<name>.php`. Assurez-vous que la classe implémente `CakeSessionHandlerInterface` et de définir Session.handler à <name>.

Pour utiliser les sessions en base de données, lancez le schéma `app/Config/Schema/sessions.php` en utilisant la commnde de shell de cake : `cake schema create Sessions`.

Security.salt Une chaîne au hasard est utilisée dans le hashage de sécurité.

Security.cipherSeed Une chaîne numérique au hasard (nombres seulement) est utilisée pour crypter/décrypter les chaînes.

Asset.timestamp Ajoute un timestamp de dernière modification du fichier particulier à la fin des urls des asset fichiers (CSS, JavaScript, Image) lors de l'utilisation de vos propres helpers. Valeurs valides : (bool) false - Ne fait rien (par défaut). (bool) true - Ajoute le timestamp quand debug > 0. (string) 'force' - Ajoute le timestamp quand debug >= 0.

Acl.classname, Acl.database Constantes utilisées pour la fonctionnalité d'Access Control List de CakePHP. Regardez le chapitre sur les Access Control Lists pour plus d'information.

Note : La configuration de mise en Cache est aussi trouvée dans core.php — Nous couvrirons cela plus tard, donc restez concentrés.

La classe `Configure` peut être utilisée pour lire et écrire des paramètres de configuration du coeur à la volée. Cela peut être spécialement pratique si vous voulez changer le paramètre de debug sur une section limitée de logique dans votre application, par exemple.

Constantes de Configuration

Alors que la plupart des options de configuration sont gérées par Configure, il y a quelques constantes que CakePHP utilise durant l'exécution.

constant `LOG_ERROR`

Constante d'Error. Utilisée pour différencier les erreurs de log et celles de debug. Actuellement PHP supporte LOG_DEBUG.

Configuration du Cache du Coeur

CakePHP utilise deux configurations de cache en interne. `_cake_model_` et `_cake_core_`. `_cake_core_` est utilisé pour stocker les chemins de fichier et les localisations d'objet. `_cake_model_` est utilisé pour stocker les descriptions de schéma, et sourcer les listes pour les sources de données. L'utilisation d'un stockage de cache rapide comme APC ou MemCached est recommandée pour ces configurations, puisqu'elles sont lues à chaque requête. Par défaut, ces eux configurations expirent toutes les 10 secondes quand le debug est supérieur à 0.

Comme toutes les données de cache sont stockées dans `Cache`, vous pouvez effacer les données en utilisant `Cache::clear()`.

Classe Configure

class `Configure`

Malgré quelques petites choses à configurer dans CakePHP, il est parfois utilie d'avoir vos propres règles de configuration pour votre application. Dans le passé, vous avoiez peut-être défini des valeurs de configuration personnalisées en définissant des variables ou des constantes dans certains fichiers. Faire cela, vous force à inclure ce fichier de configuration chaque fois que vous souhaitez utiliser ces valeurs.

La nouvelle classe Configure de CakePHP peut être utilisée pour stocker et récupèrer des valeurs spécifiques d'exécution ou d'application. Attention, cette classe vous permet de stocker tout dedans, puis de l'utiliser dans toute autre partie de votre code : une tentative évidente de casser le modèle MVC avec lequel CakePHP a été conçu. Le but principal de la classe Configure est de garder les variables centralisées qui peuvent être partagées entre beaucoup d'objets. Souvenez-vous d'essayer de suivre la règle "convention plutôt que configuration" et vous ne casserez pas la structure MVC que nous avons mis en place.

Cette classe peut être appelée de n'importe où dans l'application dans un contexte statique :

```
Configure::read('debug');
```

static `Configure::write`(*$key*, *$value*)

Paramètres

– **$key** (*string*) – La clé à écrire, peut utiliser une valeur de *notation avec points*.

– **$value** (*mixed*) – La valeur à stocker.

Utilisez `write()` pour stocker les données dans configuration de l'application :

```
Configure::write('Company.name','Pizza, Inc.');
Configure::write('Company.slogan','Pizza for your body and soul');
```

Note : La *notation avec points* utilisée dans le paramètre `$key` peut être utilisée pour organiser vos paramètres de configuration dans des groupes logiques.

L'exemple ci-dessus pourrait aussi être écrit en un appel unique :

```
Configure::write(
    'Company', array('name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and sou
);
```

Vous pouvez utiliser `Configure::write('debug', $int)` pour intervertir les modes de debug et de production à la volée. C'est particulièrement pratique pour les intéractions AMF et SOAP quand les informations de debug peuvent entraîner des problèmes de parsing.

static `Configure::read`(*$key = null*)

Paramètres

– **$key** (*string*) – La clé à lire, peut utiliser une valeur avec *notation avec points*

Utilisée pour lire les données de configuration à partir de l'application. Par défaut, la valeur de debug de CakePHP est au plus important. Si une clé est fournie, la donnée est retournée. En utilisant nos exemples du write() ci-dessus, nous pouvons lire cette donnée :

```
Configure::read('Company.name');    //yields: 'Pizza, Inc.'
Configure::read('Company.slogan');  //yields: 'Pizza for your body and soul'

Configure::read('Company');

//yields:
array('name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul');
```

Si $key est laissé à null, toutes les valeurs dans Configure seront retournées.

static `Configure::check`(*$key*)

Paramètres

– **$key** (*string*) – La clé à vérifier.

Utilisé pour vérifier si une clé/chemin existe et a une valeur non-null. Introduit dans la version 2.3 : `Configure::check()` a été ajoutée dans 2.3.

static `Configure::delete`(*$key*)

Paramètres

– **$key** (*string*) – La clé à supprimer, peut être utilisée avec une valeur en *notation avec points*

Utilisé pour supprimer l'information à partir de la configuration de l'application :

```
Configure::delete('Company.name');
```

static `Configure::version`
Retourne la version de CakePHP pour l'application courante.

static `Configure::config`(*$name*, *$reader*)

Paramètres
- **$name** (*string*) – Le nom du reader étant attaché.
- **$reader** (*ConfigReaderInterface*) – L'instance du reader étant attachée.

Attachez un reader de configuration à Configure. Les readers attachés peuvent ensuite être utilisés pour charger les fichiers de configuration. Regardez *Chargement des fichiers de configuration* pour plus d'informations sur la façon de lire les fichiers de configuration.

static `Configure::configured`(*$name = null*)

Paramètres
- **$name** (*string*) – Le nom du reader à vérifier, si null une liste de tous les readers attachés va être retournée.

Soit vérifie qu'un reader avec un nom donnée est attaché, soit récupère la liste des readers attachés.

static `Configure::drop`(*$name*)
Retire un objet reader connecté.

Lire et écrire les fichiers de configuration

CakePHP est fourni avec deux fichiers readers de configuration intégrés. `PhpReader` est capable de lire les fichiers de config de PHP, dans le même format dans lequel Configure a lu historiquement. `IniReader` est capable de lire les fichiers de config ini du coeur. Regardez la documentation PHP (http ://php.net/parse_ini_file) pour plus d'informations sur les fichiers ini spécifiés. Pour utiliser un reader de config du coeur, vous aurez besoin de l'attacher à Configure en utilisant `Configure::config()` :

```
App::uses('PhpReader', 'Configure');
// Lire les fichiers de config à partir de app/Config
Configure::config('default', new PhpReader());

// Lire les fichiers de config à partir du chemin
Configure::config('default', new PhpReader('/path/to/your/config/files/'));
```

Vous pouvez avoir de multiples readers attachés à Configure, chacun lisant différents types de fichiers de configuration, ou lisant à partir de différents types de sources. Vous pouvez intéragir avec les readers attachés en utilisant quelques autres méthodes de Configure. Pour voir, vérifier quels alias de reader sont attachés, vous pouvez utiliser `Configure::configured()` :

```
// Récupère le tableau d'alias pour les readers attachés.
Configure::configured()

// Vérifie si un reader spécifique est attaché
Configure::configured('default');
```

Vous pouvez aussi retirer les readers attachés. `Configure::drop('default')` retirerait l'alias du reader par défaut. Toute tentative future pour charger les fichiers de configuration avec ce reader serait en échec.

Chargement des fichiers de configuration

static `Configure::load`(*$key*, *$config = 'default'*, *$merge = true*)

Paramètres

- **$key** (*string*) – L'identifieur du fichier de configuration à charger.
- **$config** (*string*) – L'alias du reader configuré.
- **$merge** (*boolean*) – Si oui ou non les contenus du fichier de lecture doivent être fusionnés, ou écraser les valeurs existantes.

Une fois que vous attachez un reader de config à Configure, vous pouvez charger les fichiers de configuration :

```
// Charge my_file.php en utilisant l'objet reader 'default'.
Configure::load('my_file', 'default');
```

Les fichiers de configuration chargés fusionnent leurs données avec la configuration exécutée existante dans Configure. Cela vous permet d'écraser et d'ajouter de nouvelles valeurs dans la configuration existante exécutée. En configurant `$merge` à true, les valeurs ne vont pas toujours écraser la configuration existante.

Créer et modifier les fichiers de configuration

static `Configure::dump`(*$key*, *$config = 'default'*, *$keys = array()*)

Paramètres

- **$key** (*string*) – Le nom du fichier/configuration stockée à créer.
- **$config** (*string*) – Le nom du reader avec lequel stocker les données.
- **$keys** (*array*) – La liste des clés de haut-niveau à sauvegarder. Par défaut, pour toutes les clés.

Déverse toute ou quelques données de Configure dans un fichier ou un système de stockage supporté par le reader. Le format de sérialisation est décidé en configurant le reader de config attaché dans $config. Par exemple, si l'adaptateur 'default' est un `PhpReader`, le fichier généré sera un fichier de configuration PHP qu'on pourra charger avec `PhpReader`

Etant donné que le reader 'default' est une instance de PhpReader. Sauvegarder toutes les données de Configure dans le fichier *my_config.php* :

```
Configure::dump('my_config.php', 'default');
```

Sauvegarder seulement les erreur gérant la configuration :

```
Configure::dump('error.php', 'default', array('Error', 'Exception'));
```

`Configure::dump()` peut être utilisé pour soit modifier, soit surcharger les fichiers de configuration qui sont lisibles avec `Configure::load()` Introduit dans la version 2.2 : `Configure::dump()` a été ajouté dans 2.2.

Stocker la configuration de runtime

static `Configure::store`(*$name*, *$cacheConfig = 'default'*, *$data = null*)

Paramètres

- **$name** (*string*) – La clé de stockage pour le fichier de cache.
- **$cacheConfig** (*string*) – Le nom de la configuration de cache pour y stocker les données de configuration.
- **$data** (*mixed*) – Soit la donnée à stocker, soit laisser à null pour stocker toutes les données dans Configure.

Vous pouvez aussi stocker les valeurs de configuration exécutées pour l'utilisation dans une requête future. Depuis que configure ne se souvient seulement que des valeurs pour la requête courante, vous aurez besoin de stocker toute information de configuration modifiée si vous souhaitez l'utiliser dans des requêtes suivantes :

```
// Stocke la configuration courante dans la clé 'user_1234' dans le cache 'default'.
Configure::store('user_1234', 'default');
```

Les données de configuration stockées persistent dans la classe `Cache`. Cela vous permet de stocker les informations de Configuration dans tout moteur de stockage avec lequel `Cache` peut parler.

Restaurer la configuration de runtime

static `Configure::restore`(*$name*, *$cacheConfig = 'default'*)

Paramètres

- **$name** (*string*) – La clé de stockage à charger.
- **$cacheConfig** (*string*) – La configuration de cache à partir de laquel on charge les données.

Une fois que vous avez stocké la configuration executée, vous aurez probablement besoin de la restaurer afin que vous puissiez y accéder à nouveau. `Configure::restore()` fait exactement cela :

```
// restaure la configuration exécutée à partir du cache.
Configure::restore('user_1234', 'default');
```

Quand on restaure les informations de configuration, il est important de les restaurer avec la même clé, et la configuration de cache comme elle était utilisée pour les stocker. Les informations restaurées sont fusionnées en haut de la configuration existante exécutée.

Créer vos propres readers de Configuration

Depuis que les readers de configuration sont une partie extensible de CakePHP, vous pouvez créer des readers de configuration dans votre application et plugins. Les readers de configuration ont besoin d'implémenter l' `ConfigReaderInterface`. Cette interface définit une méthode de lecture, comme seule méthode requise. Si vous aimez vraiment les fichiers XML, vous pouvez créer un reader de config simple Xml pour votre application :

```
// dans app/Lib/Configure/XmlReader.php
App::uses('Xml', 'Utility');
class XmlReader implements ConfigReaderInterface {
    public function __construct($path = null) {
        if (!$path) {
            $path = APP . 'Config' . DS;
        }
        $this->_path = $path;
    }

    public function read($key) {
        $xml = Xml::build($this->_path . $key . '.xml');
        return Xml::toArray($xml);
    }

    // Depuis 2.3 une méthode dump() est ausi requise
    public function dump($key, $data) {
        // code pour supprimer les données d'un fichier
    }
}
```

Dans votre `app/Config/bootstrap.php`, vous pouvez attacher ce reader et l'utiliser :

```
App::uses('XmlReader', 'Configure');
Configure::config('xml', new XmlReader());
...

Configure::load('my_xml');
```

La méthode `read()` du reader de config, doit retourner un tableau d'informations de configuration que la ressource nommé `$key` contient.

interface `ConfigReaderInterface`

Définit l'interface utilisée par les classes qui lisent les données de configuration et les stocke dans `Configure`.

`ConfigReaderInterface::read`(*$key*)

Paramètres

- **$key** (*string*) – Le nom de la clé ou l'identifieur à charger.

Cette méthode devrait charger/parser les données de configuration identifiées par `$key` et retourner un tableau de données dans le fichier.

`ConfigReaderInterface::dump`(*$key*)

Paramètres

- **$key** (*string*) – L'identifieur dans lequel écrire.
- **$data** (*array*) – La donnée à supprimer.

Cette méthode doit supprimer/stocker la donnée de configuration fournie à une clé identifié par `$key`.

Introduit dans la version 2.3 : `ConfigReaderInterface::dump()` a été ajoutée dans 2.3.

exception `ConfigureException`

Lancé quand les erreurs apparaissent quand le chargement/stockage/restauration des données de configuration. Les implémentations de `ConfigReaderInterface` devraient lancer cette erreur quand elles rencontrent une erreur.

Readers de Configuration intégrés

class `PhpReader`

Vous permet de lire les fichiers de configuration qui sont stockés en fichiers PHP simples. Vous pouvez lire soit les fichiers à partir de votre `app/Config`, soit des répertoires configs du plugin en utilisant la *syntaxe de plugin*. Les fichiers **doivent** contenir une variable `$config`. Un fichier de configuration d'exemple ressemblerait à cela :

```
$config = array(
    'debug' => 0,
    'Security' => array(
        'salt' => 'its-secret'
    ),
    'Exception' => array(
        'handler' => 'ErrorHandler::handleException',
        'renderer' => 'ExceptionRenderer',
        'log' => true
    )
);
```

Des fichiers sans `$config` entraîneraient une `ConfigureException`.

Charger votre fichier de configuration personnalisé en insérant ce qui suit dans app/Config/bootstrap.php :

Configure : :load('customConfig') ;

class `IniReader`

Vous permet de lire les fichiers de configuration qui sont stockés en fichiers .ini simples. Les fichiers ini doivent être compatibles avec la fonction php `parse_ini_file`, et bénéficie des améliorations suivantes :
–Les valeurs séparées par des points sont étendues dans les tableaux.
–Les valeurs de la famille des boléens comme 'on' et 'off' sont converties en boléens.
Un fichier ini d'exemple ressemblerait à cela :

```
debug = 0

Security.salt = its-secret

[Exception]
handler = ErrorHandler::handleException
renderer = ExceptionRenderer
log = true
```

Le fichier ini ci-dessus aboutirait aux mêmes données de configuration que dans l'exemple PHP du dessus. Les structures de tableau peuvent être créées soit à travers des valeurs séparées de point, soit des sections. Les sections peuvent contenir des clés séparées de point pour des imbrications plus profondes.

Configuration de Inflection

Les conventions de nommage de Cake peuvent être vraiment sympas - vous pouvez nommer votre table de base de données big_boxes, votre model BigBox, votre controller BigBoxesController, et tout fonctionne ensemble automatiquement. La façon dont CakePHP sait comment lier les choses ensemble est en *infléctant* les mots entre leurs formes singulière et plurielle.

Il y a des occasions (spécialement pour nos amis ne parlant pas Anglais) où vous pouvez être dans des situations où l'inflecteur de CakePHP (la classe qui met au pluriel, au singulier, en CamelCase, et en underscore) ne fonctionne pas comme vous voulez. Si CakePHP ne reconnait pas vos Foci ou Fish, vous pouvez dire à CakePHP vos cas spéciaux.

Chargement d'inflections personnalisées ————————————————-

Vous pouvez utiliser `Inflector::rules()` dans le fichier `app/Config/bootstrap.php` pour charger des inflections personnalisées :

```
Inflector::rules('singular', array(
    'rules' => array('/^(bil)er$/i' => '\1', '/^(inflec|contribu)tors$/i' => '\1ta'),
    'uninflected' => array('singulars'),
    'irregular' => array('spins' => 'spinor')
));
```

ou :

```
Inflector::rules('plural', array('irregular' => array('phylum' => 'phyla')));
```

Va fusionner les règles fournies dans les ensembles d'inflection définies dans lib/Cake/Utility/Inflector.php, avec les règles ajoutées prenant le pas sur les règles du coeur.

Bootstrapping CakePHP

Si vous avez des besoins de configuration en plus, utilisez le fichier bootstrap de CakePHP dans app/Config/bootstrap.php. Ce fichier est exécuté juste après le bootstrapping du coeur de CakePHP.

Ce fichier est idéal pour un certain nombre de tâches de bootstrapping courantes :
- Définir des fonctions commodes.
- Enregistrer les constantes globales.
- Définir un model supplémentaire, une vue, et des chemins de controller.
- Créer des configurations de cache.
- Configurer les inflections.
- Charger les fichiers de configuration.

Faîtes attention de maintenir le model MVC du logiciel quand vous ajoutez des choses au fichier de bootstrap : il pourrait être tentant de placer des fonctions de formatage ici afin de les utiliser dans vos controllers.

Résister à la tentation. Vous serez content plus tard d'avoir suivi cette ligne de conduite.

Vous pouvez aussi envisager de placer des choses dans la classe `AppController`. Cette class est une classe parente pour tous les controllers dans votre application. `AppController` est un endroit pratique pour utiliser les callbacks de controller et définir des méthodes à utiliser pour tous les controllers.

Routing

Routing est une fonctionnalité qui mappe les URLs aux actions du controller. Elle a été ajoutée à CakePHP pour rendre les URLs belles et plus configurables et flexibles. L'utilisation du mod_rewrite de Apache n'est pas nécessaire pour utiliser les routes, mais cela rendra votre barre d'adresse beaucoup plus élégante.

Le Routing dans CakePHP englobe aussi l'idée de routing inversé, où un tableau de paramètres peut être inversé en une chaîne url. En utilisant le routing inversé, vous pouvez facilement reconstruire votre structure d'url des applications sans avoir mis à jour tous vos codes.

Configuration des Routes

Les Routes dans une application sont configurées dans `app/Config/routes.php`. Ce fichier est inclu par le `Dispatcher` quand on gère les routes et vous permet de définir des routes spécifiques d'application que vous voulez utiliser. Les Routes déclarées dans ce fichier sont traitées de haut en bas quand les requêtes entrantes correspondent. Cela signifie que l'ordre dans lequel vous placez les routes peuvent affecter comment les routes sont parsées. C'est généralement une bonne idée de placer les routes visitées le plus fréquemment en haut du fichier de routes si possible. Cela va permettre de ne pas à avoir à vérifier un certain nombre de routes qui ne correspondront pas à chaque requête.

Les Routes sont parsées et matchées dans l'ordre dans lequel elles sont connectées. Si vous définissez deux routes similaires, la première route définie va avoir une priorité plus haute sur celle définie plus tard. Après avoir connecté les routes, vous pouvez manipuler l'ordre des routes en utilisant `Router::promote()`.

CakePHP vient aussi avec quelques routes par défaut pour commencer. Celles-ci peuvent être désactivées plus tard une fois que vous êtes sûr que vous n'en aurez pas besoin. Regardez *Désactiver les routes par défaut* sur la façon de désactiver le routing par défaut.

Routing par défaut

Avant que vous appreniez à configurer vos propres routes, vous devez savoir que CakePHP est configuré avec un ensemble de routes par défaut. Le routing de CakePHP par défaut va vous faire aller assez loin dans toute application. Vous pouvez accéder à une action directement par l'URL en mettant son nom dans la requête. Vous pouvez aussi passer des paramètres aux actions de votre controller en utilisant l'URL. :

```
modèle URL des routes par défaut:
http://example.com/controller/action/param1/param2/param3
```

L'URL /posts/view mappe à l'action view() de PostsController, et /products/view_clearance mappe vers l'action view_clearance() de ProductsController. Si aucune action n'est spécifiée dans l'URL, la méthode index() est supposée.

La configuration du routing par défaut vous permet aussi de passer les paramètres à vos actions en utilisant l'URL. Une requête pour /posts/view/25 serait équivalente à appeler view(25) dans le PostsController, par exemple. Le routing par défaut fournit aussi les routes pour les plugins, et les routes préfixées si vous choisissez d'utiliser ces fonctionnalités.

Les routes intégrées sont dans `Cake/Config/routes.php`. Vous pouvez désactiver le routing par défaut en les retirant du fichier *routes.php* de votre application.

Connecter les Routes

Définir vos propres routes vous permet de définir la façon dont votre application va répondre à une URL donnée. Définir vos propres routes dans le fichier `app/Config/routes.php` en utilisant la méthode `Router::connect()`.

La méthode `connect()` prend trois paramètres : l'URL que vous souhaitez faire correspondre, les valeurs par défaut pour les éléments de votre route, et les règles d'expression régulière pour aider le routeur à faire correspondre les éléments dans l'URL.

Le format basique pour une définition de route est :

```
Router::connect(
    'URL',
    array('default' => 'defaultValue'),
    array('option' => 'matchingRegex')
);
```

Le premier paramètre est utilisé pour dire au routeur quelle sorte d'URL vous essayez de contrôler. L'URL est une chaîne normale délimitée par des slashes, mais peut aussi contenir une wildcard (*) ou *Les éléments de Route*. Utiliser une wildcard dit au routeur que vous êtes prêt à accepter tout argument supplémentaire fourni. Les Routes sans un * ne matchent que le pattern template exact fourni.

Une fois que vous spécifiez une URL, vous utilisez les deux derniers paramètres de `connect()` pour dire à CakePHP quoi faire avec une requête une fois qu'elle a été matchée. Le deuxième paramètre est un tableau associatif. Les clés du tableau devraient être appelées après les éléments de route dans l'URL, ou les éléments par défaut : `:controller`, `:action`, et `:plugin`. Les valeurs dans le tableau sont les valeurs par défaut pour ces clés. Regardons quelques exemples simples avant que nous commencions l'utilisation le troisième paramètre de connect() :

```
Router::connect(
    '/pages/*',
    array('controller' => 'pages', 'action' => 'display')
);
```

Cette route est trouvée dans le fichier routes.php distribué avec CakePHP. Cette route matche toute URL commençant par `/pages/` et il tend vers l'action `display()` de `PagesController();` La requête /pages/products serait mappé vers `PagesController->display('products')`.

En plus de l'étoile greedy `/*` il y aussi la syntaxe de l'étoile trailing `/**`. Utiliser une étoile double trailing, va capturer le reste de l'URL en tant qu'argument unique passé. Ceci est utile quand vous voulez utilisez un argument qui incluait un `/` dedans :

```
Router::connect(
    '/pages/**',
    array('controller' => 'pages', 'action' => 'show')
);
```

L'URL entrante de `/pages/the-example-/-and-proof` résulterait en un argument unique passé de `the-example-/-and-proof`. Introduit dans la version 2.1. Vous pouvez utiliser le deuxième paramètre de `Router::connect()` pour fournir tout paramètre de routing qui est composé des valeurs par défaut de la route :

```
Router::connect(
    '/government',
    array('controller' => 'products', 'action' => 'display', 5)
);
```

Cet exemple montre comment vous pouvez utilisez le deuxième paramètre de `connect()` pour définir les paramètres par défaut. Si vous construisez un site qui propose des produits pour différentes catégories de clients, vous pourriez considérer la création d'une route. Cela vous permet de vous lier à `/government` plutôt qu'à `/pages/display/5`.

Note : Bien que vous puissiez connecter des routes alternatives, les routes par défaut vont continuer à fonctionner. Ceci pourrait créer des situations, où le contenu pourrait finir avec 2 urls. Regardez *Désactiver les routes par défaut* pour désactiver les routes par défaut, et fournir seulement les urls que vous définissez.

Une autre utilisation ordinaire pour le Router est de définir un "alias" pour un controller. Disons qu'au lieu d'accéder à notre URL régulière à `/users/some_action/5`, nous aimerions être capable de l'accéder avec `/cooks/some_action/5`. La route suivante s'occupe facilement de cela :

```
Router::connect(
    '/cooks/:action/*', array('controller' => 'users')
);
```

Cela dit au Router que toute url commençant par `/cooks/` devrait être envoyée au controller users. L'action appelée dépendra de la valeur du paramètre `:action`. En utilisant *Les éléments de Route*, vous pouvez créer des routes variables, qui acceptent les entrées utilisateur ou les variables. La route ci-dessus utilise aussi l'étoile greedy. L'étoile greedy indique au `Router` que cette route devrait accepter tout argument de position supplémentaire donné. Ces arguments seront rendus disponibles dans le tableau *Arguments passés*.

Quand on génére les urls, les routes sont aussi utilisées. Utiliser `array('controller' => 'users', 'action' => 'some_action', 5)` en url va sortir /cooks/some_action/5 si la route ci-dessus est la première correspondante trouvée.

Si vous pensez utiliser des arguments nommés personnalisés avec votre route, vous devrez avertir le routeur de cela en utilisant la fonction `Router::connectNamed()`. Donc si vous voulez que la route ci-dessus matchent les urls comme `/cooks/some_action/type:chef`, nous faisons :

```
Router::connectNamed(array('type'));
Router::connect(
    '/cooks/:action/*', array('controller' => 'users')
);
```

Les éléments de Route

Vous pouvez spécifier vos propres éléments de route et ce faisant cela vous donne le pouvoir de définir des places dans l'URL où les paramètres pour les actions du controller doivent reposer. Quand une requête est faite, les valeurs pour ces éléments de route sont trouvées dans `$this->request->params` dans le controller. Ceci est différent de la façon dont les paramètres sont gérés, donc notez la différence : les paramètres nommés (/controller/action/name :value) sont trouvés dans `$this->request->params['named']`, alors que la donnée de l'élément de route personnalisé est trouvé dans `$this->request->params`.

quand vous définissez un élément de route personnalisé, vous pouvez spécifier en option une expression régulière - cela dit à CakePHP comment savoir si l'URL est correctement formée ou non. Si vous choisissez de ne pas fournir une expression régulière, toute expression non `/` sera traitée comme une partie du paramètre :

```
Router::connect(
    '/:controller/:id',
    array('action' => 'view'),
    array('id' => '[0-9]+')
);
```

Cet exemple simple montre comment créer une manière rapide de voir les models à partir de tout controller en élaborant une URL qui ressemble à `/controllername/:id`. L'URL fourni à connect() spécifie deux éléments de route : `:controller` et `:id`. L'élément `:controller` est l'élément de route par défaut de CakePHP, donc le routeur sait comment matcher et identifier les noms de controller dans les URLs. L'élément `:id` est un élément de route personnalisé, et doit être clarifié plus loin en spécifiant une expression régulière correspondante dans le troisième paramètre de connect().

Note : Les Patrons utilisés pour les éléments de route ne doivent pas contenir de groupes capturés. Si ils le font, le Router ne va pas fonctionner correctement.

Une fois que cette route a été définie, requêtant `/apples/5` est la même que celle requêtant `/apples/view/5`. Les deux appeleraient la méthode view() de ApplesController. A l'intérieur de la méthode view(), vous aurez besoin d'accéder à l'ID passé à `$this->request->params['id']`.

Si vous avez un unique controller dans votre application et que vous ne ne voulez pas que le nom du controller apparaisse dans l'url, vous pouvez mapper tous les urls aux actions dans votre controller. Par exemple, pour mapper toutes les urls aux actions du controller `home`, par ex avoir des urls comme `/demo` à la place de `/home/demo`, vous pouvez faire ce qui suit :

```
Router::connect('/:action', array('controller' => 'home'));
```

Si vous souhaitez fournir une url non sensible à la casse, vous pouvez utiliser les modificateurs en ligne d'expression régulière :

```
Router::connect(
    '/:userShortcut',
    array('controller' => 'teachers', 'action' => 'profile', 1),
    array('userShortcut' => '(?i:principal)')
);
```

Un exemple de plus, et vous serez un routing pro :

```
Router::connect(
    '/:controller/:year/:month/:day',
    array('action' => 'index'),
    array(
        'year' => '[12][0-9]{3}',
        'month' => '0[1-9]|1[012]',
        'day' => '0[1-9]|[12][0-9]|3[01]'
```

```
    )
);
```

C'est assez complexe, mais montre comme les routes peuvent vraiment devenir puissantes. L'URL fourni a quatre éléments de route. Le premier nous est familier : c'est une route par défaut qui dit à CakePHP d'attendre un nom de controller.

Ensuite, nous spécifions quelques valeurs par défaut. Quelque soit le controller, nous voulons que l'action index() soit appelée. Nous définissons le paramètre jour (le quatrième élément dans l'URL) à null pour le marquer en option.

Finalement, nous spécifions quelques expressions régulières qui vont matcher les années, mois et jours sous forme numérique. Notez que les parenthèses (le groupement) ne sont pas supportées dans les expressions régulières. Vous pouvez toujours spécifier des alternatives, comme dessus, mais ne pas grouper avec les parenthèses.

Une fois définie, cette route va matcher `/articles/2007/02/01`, `/posts/2004/11/16`, gérant les requêtes pour les actions index() de ses controllers respectifs, avec les paramètres de date dans `$this->request->params`.

Il y a plusieurs éléments de route qui ont une signification spéciale dans CakePHP, et ne devraient pas être utilisés à moins que vous souhaitiez spécifiquement la signification.
- `controller` Utilisé pour nommer le controller pour une route.
- `action` Utilisé pour nommer l'action de controller pour une route.
- `plugin` Utilisé pour nommer le plugin dans lequel un controller est localisé.
- `prefix` Utilisé pour *Prefix de routage*.
- `ext` Utilisé pour le routing *Extensions de Fichier*.

Passer des paramètres à l'action

Quand vous connectez les routes en utilisant *Les éléments de Route* vous voudrez peut-être que des éléments routés soient passés aux arguments à la place. En utilisant le 3ème argument de `Router::connect()`, vous pouvez définir quels éléments de route doivent aussi être rendus disponibles en arguments passés :

```
// SomeController.php
public function view($articleId = null, $slug = null) {
    // du code ici...
}

// routes.php
Router::connect(
    '/blog/:id-:slug', // E.g. /blog/3-CakePHP_Rocks
    array('controller' => 'blog', 'action' => 'view'),
    array(
        // order matters since this will simply map ":id" to $articleId in your action
        'pass' => array('id', 'slug'),
        'id' => '[0-9]+'
    )
);
```

et maintenant, grâce aux possibilités de routing inversé, vous pouvez passer dans le tableau d'url comme ci-dessous et Cake sait comment former l'URL comme définie dans les routes :

```
// view.ctp
// cela va retourner un lien vers /blog/3-CakePHP_Rocks
echo $this->Html->link('CakePHP Rocks', array(
    'controller' => 'blog',
    'action' => 'view',
    'id' => 3,
    'slug' => 'CakePHP_Rocks'
));
```

Paramètres nommées Per-route

Alors que vous pouvez contrôler les paramètres nommés à une grande échelle en utilisant `Router::connectNamed()`, vous pouvez aussi contrôler le comportement des paramètres nommés au niveau de la route en utilisant le 3ème argument de `Router::connect()` :

```
Router::connect(
    '/:controller/:action/*',
    array(),
    array(
        'named' => array(
            'wibble',
            'fish' => array('action' => 'index'),
            'fizz' => array('controller' => array('comments', 'other')),
            'buzz' => 'val-[\d]+'
        )
    )
);
```

La définition de la route ci-dessus utilise la clé `named` pour définir comment plusieurs paramètres nommés devraient être traitées. Regardons chacune des différentes règles une par une :

- 'wibble' n'a pas d'information en plus. Cela signifie qu'il va toujours parser si il est trouvé dans une url matchant cette route.
- 'fish' a un tableau de conditions, contenant la clé 'action'. Cela signifie que fish va être seulement parsé en paramètre nommé si l'action est aussi indicée.
- 'fizz' a aussi un tableau de conditions. Cependant, il contient deux controllers, cela signifie que 'fizz' va seulement être parsé si le controller matche un des noms dans le tableau.
- 'buzz' a une condition de type chaîne de caractères. Les conditions en chaîne sont traitées comme des fragments d'expression régulière. Seules les valeurs pour buzz matchant le pattern vont être parsées.

Si un paramètre nommé est utilisé et qu'il ne matche pas le critère fourni, il sera traité comme un argument passé au lieu d'un paramètre nommé.

Prefix de routage

De nombreuses applications nécessitent une section d'administration dans laquelle les utilisateurs privilégiés peuvent faire des modifications. Ceci est souvent réalisé grâce à une URL spéciale telle que `/admin/users/edit/5`. Dans CakePHP, les préfixes de routage peuvent être activés depuis le fichier

de configuration du cœur en configurant les préfixes avec Routing.prefixes. Notez que les prefixes, bien que liés au routeur sont configurés dans `app/Config/core.php` :

```
Configure::write('Routing.prefixes', array('admin'));
```

Dans votre controller, toute action avec le préfixe `admin_` sera appelée. En utilisant notre exemple des users, accéder à l'url `/admin/users/edit/5` devrait appeler la méthode `admin_edit` de notre `UsersController` en passant 5 comme premier paramètre. Le fichier de vue correspondant devra être `app/View/Users/admin\_edit.ctp`.

Vous pouvez faire correspondre l'url /admin à votre action `admin_index` du controller Pages en utilisant la route suivante :

```
Router::connect('/admin', array('controller' => 'pages', 'action' => 'index', 'admin' => tr
```

Vous pouvez aussi configurer le Router pour utiliser plusieurs préfixes. En ajoutant des valeurs supplémentaires dans `Routing.prefixes`. Si vous définissez :

```
Configure::write('Routing.prefixes', array('admin', 'manager'));
```

Cake va automatiquement générer les routes pour les deux prefixes admin et manager. Chaque préfixe configuré va avoir les routes générées suivantes pour cela :

```
Router::connect("/{$prefix}/:plugin/:controller", array('action' => 'index', 'prefix' => $p
Router::connect("/{$prefix}/:plugin/:controller/:action/*", array('prefix' => $prefix, $pre
Router::connect("/{$prefix}/:controller", array('action' => 'index', 'prefix' => $prefix, $
Router::connect("/{$prefix}/:controller/:action/*", array('prefix' => $prefix, $prefix => t
```

Un peu comme le routing admin, toutes les actions préfixées doivent être préfixées avec le nom du préfixe. Ainsi `/manager/posts/add` map vers `PostsController::manager_add()`.

De plus, le préfixe courant sera disponible à partir des méthodes du controller avec `$this->request->prefix`

Quand on utilise les routes préfixées, il est important de se rappeler qu'en utilisant le helper HTML pour construire vos liens va aider à maintenir les appels préfixés. Voici comment construire le lien en utilisant le helper HMTL :

```
// Allez dans une route préfixée.
echo $html->link('Manage posts', array('manager' => true, 'controller' => 'posts', 'action'

// laissez un préfixe
echo $html->link('View Post', array('manager' => false, 'controller' => 'posts', 'action' =
```

Routing des Plugins

Le routage des Plugins utilise la clé **plugin**. Vous pouvez créer des liens qui pointent vers un plugin, mais en ajoutant la clé plugin à votre tableau d'url :

```
echo $html->link('New todo', array('plugin' => 'todo', 'controller' => 'todo_items', 'actio
```

Inversement, si la requête active est une requête de plugin et que vous voulez créer un lien qui ne pointe pas vers un plugin, vous pouvez faire ce qui suit :

```
echo $html->link('New todo', array('plugin' => null, 'controller' => 'users', 'action' =>
```

En définissant `plugin => null`, vous indiquez au Routeur que vous souhaitez créer un lien qui n'est pas une partie d'un plugin.

Extensions de Fichier

Pour manipuler différentes extensions de fichier avec vos routes, vous avez besoin d'une ligne supplémentaire dans votre fichier de config des routes :

```
Router::parseExtensions('html', 'rss');
```

Ceci indiquera au routeur de supprimer toutes extensions de fichiers correspondantes et ensuite d'analyser ce qui reste.

Si vous voulez créer une URL comme /page/titre-de-page.html, vous devriez créer votre route comme illustré ci-dessous :

```
Router::connect(
    '/page/:title',
    array('controller' => 'pages', 'action' => 'view'),
    array(
        'pass' => array('title')
    )
);
```

Ensuite pour créer des liens qui s'adapteront aux routes, utilisez simplement :

```
$html->link(
    'Link title',
    array('controller' => 'pages', 'action' => 'view', 'title' => 'super-article', 'ext' =>
);
```

Les extensions de Fichier sont utilisées par `RequestHandlerComponent` pour faire automatiquement le changement de vue basé sur les types de contenu. Regardez RequestHandlerComponent pour plus d'informations.

Utiliser des conditions supplémentaires de correspondance des routes

Quand vous créez des routes, vous souhaitez restreindre certaines URL basées sur des configurations requête/environnement spécifique. Un bon exemple de cela est le routing *REST*. Vous pouvez spécifier des conditions supplémentaires dans l'argument `$defaults` pour `Router::connect()`. Par défaut, CakePHP propose 3 conditions d'environment, mais vous pouvez en ajouter plus en utilisant *Classes de Route personnalisées*. Les options intégrées sont :

- `[type]` Seulement les requêtes correspondantes pour des types de contenu spécifiques.
- `[method]` Seulement les requêtes correspondantes avec des verbes HTTP spécifiques.

– `[server]` Correspond seuelement quand $_SERVER['SERVER_NAME'] correspond à la valeur donnée.

Nous allons fournir un exemple simple ici pour montrer comment vous pouvez utiliser l'options `[method]` pour créer une route Restful personnalisée :

```
Router::connect(
    "/:controller/:id",
    array("action" => "edit", "[method]" => "PUT"),
    array("id" => "[0-9]+")
);
```

La route ci-dessus va seulement correspondre aux requêtes `PUT`. En utilisant ces conditions, vous pouvez créer un routing REST personnalisé, ou d'autres requêtes de données dépendant d'information.

Arguments passés

Les arguments passés sont des arguments supplémentaires ou des segments du chemin qui sont utilisés lors d'une requête. Ils sont souvent utilisés pour transmettre des paramètres aux méthodes de vos controllers. :

```
http://localhost/calendars/view/recent/mark
```

Dans l'exemple ci-dessus, `recent` et `mark` tous deux des arguments passés à `CalendarsController::view()`. Les arguments passés sont transmis aux contrôleurs de trois manières. D'abord comme arguments de la méthode de l'action appelée, deuxièmement en étant accessibles dans `$this->request->params['pass']` sous la forme d'un tableau indexé numériquement. Enfin, il y a `$this->passedArgs` disponible de la même façon que la deuxième façon. Lorsque vous utilisez des routes personnalisées il est possible de forcer des paramètres particuliers comme étant des paramètres passés également. Voir passer des paramètres à une action pour plus d'informations.

Si vous alliez visiter l'url mentionné précédemment, et que vous aviez une action de controller qui ressemblait à cela :

```
CalendarsController extends AppController{
    public function view($arg1, $arg2) {
        debug(func_get_args());
    }
}
```

Vous auriez la sortie suivante :

```
Array
(
    [0] => recent
    [1] => mark
)
```

La même donnée est aussi disponible dans `$this->request->params['pass']` et dans `$this->passedArgs` dans vos controllers, vues, et helpers. Les valeurs dans le tableau pass sont indicées numériquement basé sur l'ordre dans lequel elles apparaissent dans l'url appelé :

```
debug($this->request->params['pass']);
debug($this->passedArgs);
```

Les deux du dessus sortiraient :

```
Array
(
    [0] => recent
    [1] => mark
)
```

Note : $this->passedArgs peut aussi contenir des paramètres nommés comme jun tableau mixte nommé avec des arguments passés.

Quand vous générez des urls, en utilisant un *tableau routing*, vous ajoutez des arguments passés en valeurs sans clés de type chaîne dans le tableau :

```
array('controller' => 'posts', 'action' => 'view', 5)
```

Comme `5` a une clé numérique, il est traité comme un argument passé.

Paramètres nommés

Vous pouvez nommer les paramètres et envoyer leurs valeurs en utilisant l'URL. Une requête pour `/posts/view/title:first/category:general` résultera en un appel à l'action view() du controller PostsController. Dans cette action, vous trouverez les valeurs des paramètres "title" et "category" dans `$this->params['named']`. Vous pouvez également accéder aux paramètres nommés depuis `$this->passedArgs`. Dans les deux cas, vous pouvez accéder aux paramètres nommés en utilisant leur nom en index. Si les paramètres nommés sont omis, ils ne seront pas définis.

Quelques exemples de routes par défaut seront plus parlants.

Note : Ce qui est parsé en paramètre nommé est contrôlé par `Router::connectNamed()`. Si vos paramètres nommés ne sont pas du routing inversé, ou ne sont pas parsés correctement, vous aurez besoin d'informer `Router` sur eux.

Quelques exemples pour résumer les routes par défaut peuvent prouver leur aide :

```
URL vers le mapping de l'action du controller utilisant les routes par
défaut:

URL: /monkeys/jump
Mapping: MonkeysController->jump();

URL: /products
Mapping: ProductsController->index();

URL: /tasks/view/45
```

```
Mapping: TasksController->view(45);

URL: /donations/view/recent/2001
Mapping: DonationsController->view('recent', '2001');

URL: /contents/view/chapter:models/section:associations
Mapping: ContentsController->view();
$this->passedArgs['chapter'] = 'models';
$this->passedArgs['section'] = 'associations';
$this->params['named']['chapter'] = 'models';
$this->params['named']['section'] = 'associations';
```

Lorsque l'on fait des routes personnalisées, un piège classique est d'utiliser des paramètres nommés qui casseront vos routes. Pour résoudre cela vous devez informer le Router des paramètres qui sont censés être des paramètres nommés. Sans cette information, le Routeur est incapable de déterminer si les paramètres nommés doivent en effet être des paramètres nommés ou des paramètres à router, et supposera par défaut que ce sont des paramètres à router. Pour connecter des paramètres nommés dans le routeur utilisez `Router::connectNamed()` :

```
Router::connectNamed(array('chapter', 'section'));
```

Va s'assurer que votre chapitre et les paramètres de section inversent les routes correctement.

Quand vous générez les urls, en utilisant un *tableau routing*, vous ajoutez les paramètres nommés en valeurs avec les clés en chaîne matchant le nom :

```
array('controller' => 'posts', 'action' => 'view', 'chapter' => 'association')
```

Puisque 'chapter' ne matche aucun élément de route défini, il est traité en paramètre nommé.

Note : Les deux paramètres nommés et les éléments de route partagent le même espace-clé. Il est mieux d'éviter de réutiliser une clé pour les deux, élément de route et paramètre nommé.

Les paramètres nommés supportent aussi l'utilisation de tableaux pour générer et parser les urls. La syntaxe fonctionne de façon très similaire à la syntaxe de tableau utilisée pour les paramètres GET. Quand vous générez les urls, vous pouvez utiliser la syntaxe suivante :

```
$url = Router::url(array(
    'controller' => 'posts',
    'action' => 'index',
    'filter' => array(
        'published' => 1
        'frontpage' => 1
    )
));
```

Ce qui est au-dessus générerait l'url `/posts/index/filter[published]:1/filter[frontpage]:1`. Les paramètres sont ensuite parsés et stockés dans la variable passedArgs de votre controller en tableau, de la même façon que vous les envoyez au `Router::url` :

```
$this->passedArgs['filter'] = array(
    'published' => 1
    'frontpage' => 1
);
```

Les tableaux peuvent aussi être imbriqués en profondeur, vous autorisant même à plus de flexibilité dans les arguments passés :

```
$url = Router::url(array(
    'controller' => 'posts',
    'action' => 'search',
    'models' => array(
        'post' => array(
            'order' => 'asc',
            'filter' => array(
                'published' => 1
            )
        ),
        'comment' => array(
            'order' => 'desc',
            'filter' => array(
                'spam' => 0
            )
        ),
    ),
    'users' => array(1, 2, 3)
));
```

Vous finiriez avec une longue et belle url comme ceci (entouré pour une lecture facile) :

```
posts/search
  /models[post][order]:asc/models[post][filter][published]:1
  /models[comment][order]:desc/models[comment][filter][spam]:0
  /users[]:1/users[]:2/users[]:3
```

Et le tableau résultant qui serait passé au controller matcherait ceci que vous avez passé au routeur :

```
$this->passedArgs['models'] = array(
    'post' => array(
        'order' => 'asc',
        'filter' => array(
            'published' => 1
        )
    ),
    'comment' => array(
        'order' => 'desc',
        'filter' => array(
            'spam' => 0
        )
    ),
);
```

Contrôler les paramètres nommés

Vous pouvez contrôler la configuration du paramètre nommé au niveau-par-route ou les contrôler globalement. Le contrôle global est fait à travers `Router::connectNamed()`. Ce qui suit donne quelques exemples de la façon dont vous contrôlez le parsing du paramètre nommé avec connectNamed().

Ne parsez aucun paramètre nommé :

```
Router::connectNamed(false);
```

Parsez seulement les paramètres par défaut utilisés pour la pagination de CakePHP :

```
Router::connectNamed(false, array('default' => true));
```

Parsez seulement le paramètre de la page si sa valeur est un nombre :

```
Router::connectNamed(array('page' => '[\d]+'), array('default' => false, 'greedy' => false
```

Parsez seulement le paramètre de la page dans tous les cas :

```
Router::connectNamed(array('page'), array('default' => false, 'greedy' => false));
```

Parsez seulement le paramètre de la page si l'action courante est 'index' :

```
Router::connectNamed(
    array('page' => array('action' => 'index')),
    array('default' => false, 'greedy' => false)
);
```

Parsez seulement le paramètre de la page si l'action courante est 'index' et le controller est 'pages' :

```
Router::connectNamed(
    array('page' => array('action' => 'index', 'controller' => 'pages')),
    array('default' => false, 'greedy' => false)
);
```

connectNamed() supporte un certain nombre d'options :

- `greedy` Configurer cela à true fera que le Router va parser tous les paramètres nommés. Configurer cela à false va parser seulement les paramètres nommés.
- `default` Définissez cela à true pour fusionner dans l'ensemble par défaut des paramètres nommés.
- `reset` Définissez à true pour effacer les règles existantes et recommencer à zéro.
- `separator` Changez la chaîne utilisée pour séparer la clé & valeur dans un paramètre nommé. Par défaut *:*

Routing inversé

Le routing inversé est une fonctionnalité dans CakePHP qui est utilisée pour vous permettre de changer facilement votre structure d'url sans avoir à modifier tout votre code. En utilisant *routing arrays* pour définir vos urls, vous pouvez configurer les routes plus tard et les urls générés vont automatiquement être mises à jour.

Si vous créez des urls en utilisant des chaînes de caractères comme :

```
$this->Html->link('View', '/posts/view/' + $id);
```

Et ensuite plus tard, vous décidez que `/posts` devrait vraiment être appelé 'articles' à la place, vous devrez aller dans toute votre application en renommant les urls. Cependant, si vous définissiez votre lien comme :

```
$this->Html->link(
    'View',
    array('controller' => 'posts', 'action' => 'view', $id)
);
```

Ensuite quand vous décidez de changer vos urls, vous pouvez le faire en définissant une route. Cela changerait à la fois le mapping d'URL entrant, ainsi que les urls générés.

Quand vous utilisez les urls en tableau, vous pouvez définir les paramètres chaîne de la requête et les fragments de document en utilisant les clés spéciales :

```
Router::url(array(
    'controller' => 'posts',
    'action' => 'index',
    '?' => array('page' => 1),
    '#' => 'top'
));

// va générer une url comme.
/posts/index?page=1#top
```

Routing inversé

Rediriger le routing vous permet de délivrer des redirections à l'état HTTP 30x pour les routes entrantes, et les pointent aux différentes urls. Ceci est utilisé quand vous voulez informer les applications clientes qu'une ressource a été déplacée et que vous ne voulez pas avoir deux urls pour le même contenu.

Les routes de redirection sont différentes des routes normales puisqu'elles effectuent une redirection du header actuel si une correspondance est trouvée. La redirection peut survenir vers une destination dans votre application ou une localisation en-dehors :

```
Router::redirect(
    '/home/*',
    array('controller' => 'posts', 'action' => 'view',
    array('persist' => true) // ou array('persist'=>array('id')) pour un routing par défau
);
```

Redirige `/home/*` vers `/posts/view` et passe les paramètres vers `/posts/view`. Utiliser un tableau en une destination de redirection vous permet d'utiliser d'autres routes pour définir où une chaîne url devrait être redirigée. Vous pouvez rediriger vers des localisations externes en utilisant les chaînes url en destination :

```
Router::redirect('/posts/*', 'http://google.com', array('status' => 302));
```

Cela redirigerait `/posts/*` vers `http://google.com` avec un état statut HTTP à 302.

Désactiver les routes par défaut

Si vous avez complètement personnalisé toutes les routes, et voulez éviter toute pénalité de contenu dupliqué possible des moteurs de recherche, vous pouvez retirer les routes par défaut que CakePHP offre en les supprimant de votre fichier d'application routes.php.

Cela fera en sorte que CakePHP serve les erreurs, quand les utilisateurs essaient de visiter les urls qui seraient normalement fournies par CakePHP mais n'ont pas été connectée explicitement.

Classes de Route personnalisées

Les classes de route personnalisées vous permettent d'étendre et de modifier la façon dont certaines routes parsent les demandes et de traiter le routing inversé. Une classe de la route devrait hériter de la classe `CakeRoute` et mettre en œuvre un ou les deux `match()` et/ou `parse()`. `parse()` est utilisée pour analyser les demandes et correspondance et `match()` est utilisée pour traiter les routes inversées.

Vous pouvez utiliser une classe de route personnalisée lors d'un création d'une route à l'aide des options de la classe `routeClass`, et en chargeant le fichier contenant votre routes avant d'essayer de l'utiliser :

```
Router::connect(
     '/:slug',
     array('controller' => 'posts', 'action' => 'view'),
     array('routeClass' => 'SlugRoute')
);
```

Cette route créerait une instance de `SlugRoute` et vous permet d'implémenter la gestion de paramètre personnalisée.

API du Router

class `Router`

Le Router gère la génération des urls sortants, et le parsing de la requête url entrante dans les ensembles de paramètre que CakePHP peut dispatcher.

static `Router::connect` (*$route*, *$defaults* = *array()*, *$options* = *array()*)

Paramètres

- **$route** (*string*) – Une chaîne décrivant le template de la route.
- **$defaults** (*array*) – Un tableau décrivant les paramètres de la route par défaut. Ces paramètres seront utilisés par défaut et peuvent fournir des paramètres de routing qui ne sont pas dynamiques.
- **$options** (*array*) – Un tableau matchant les éléments nommés dans la route aux expressions régulières avec lesquels cet élément devrait correspondre. Contient aussi des paramètres supplémentaires comme les paramètres routés doivent être passés dans les arguments passés, en fournissant les patterns pour les paramètres de routing et fournir le nom d'une classe de routing personnalisée.

Les routes ont une façon de connecter les requêtes urls aux objets dans votre application. Dans les routes du coeur, il y a un ensemble d'expressions régulières qui sont utilisées pour matcher les requêtes aux destinations.

Exemples :

```
Router::connect('/:controller/:action/*');
```

Le premier paramètre va être utilisé comme nom de controller alors que le second est utilisé en nom d'action. La syntaxe '/*' rend cette route greedy puisqu'elle ca matcher les requêtes comme */posts/index* ainsi que les requêtes comme `/posts/edit/1/foo/bar`.:

```
Router::connect('/home-page', array('controller' => 'pages', 'action' => 'display', 'h
```

Ce qui est au-dessus montre l'utilisation d'un paramètre de route par défaut. Et fournit les paramètres de routing pour une route statique. :

```
Router::connect(
    '/:lang/:controller/:action/:id',
    array(),
    array('id' => '[0-9]+', 'lang' => '[a-z]{3}')
);
```

Montre la connexion d'une route avec les paramètres de route personnalisé ainsi que fournit les patterns pour ces paramètres. Les patterns pour les paramètres de routing n'ont pas besoin de capturer les groupes, puisque l'un d'eux sera ajouté pour chaque paramètre de route.

$options propose trois clés 'special'. `pass`, `persist` et `routeClass` ont une signification spéciale dans le tableau $options.

- `pass` est utilisé pour définir lesquels des paramètres routés devrait être passé dans le tableau pass. Ajouter un paramètre à pass le retirera du tableau de route régulière. Ex. `'pass' => array('slug')`.
- `persist` est utilisé pour définir lesquels des paramètres de route devrait être automatiquement inclus quand on génére les nouvels urls. Vous pouvez écraser les paramètres persistentes en les redéfinissant dans une url ou les retirer en configurant le paramètre à `false`. Ex. `'persist' => array('lang')`.
- `routeClass` est utilisé pour étendre et changer la façon dont les routes individuelles parsent les requêtes et gèrent le routing inversé, via une classe de routing personnalisée. Ex. `'routeClass' => 'SlugRoute'`.
- `named` est utilisé pour configurer les paramètres nommés au niveau de la route. Cette clé utilise les mêmes options que `Router::connectNamed()`.

static `Router::redirect` (*$route*, *$url*, *$options* = *array()*)

Paramètres

- **$route** (*string*) – Un template de route qui dicte quels urls devraient être redirigées.
- **$url** (*mixed*) – Soit un *tableau routing*, soit une chaîne url pour la destination du redirect.
- **$options** (*array*) – Un tableau d'options pour le redirect.

Connecte une nouvelle redirection de Route dans le routeur. Regardez *Routing inversé* pour plus d'informations.

static `Router::connectNamed` (*$named*, *$options* = *array()*)

Paramètres

- **$named** (*array*) – Une liste des paramètres nommés. Les paires de valeur clé sont acceptées où les valeurs sont soit des chaînes regex à matcher, soit des tableaux.

– **$options** (*array*) – Permet le contrôle de toutes les configurations : separator, greedy, reset, default.

Spécifie quels paramètres nommés CakePHP devrait parsés en urls entrantes Par défaut, CakePHP va parser tout paramètre nommé en-dehors des URLS entrantes. Regardez *Contrôler les paramètres nommés* pour plus d'informations.

static `Router::promote`(*$which = null*)

Paramètres

– **$which** (*integer*) – Un indice de tableau à 0 représentant la route à déplacer. Par exemple, si 3 routes ont été ajoutée, la dernière route serait 2.

Favorise une route (par défaut, le dernier ajouté) au début de la liste.

static `Router::url`(*$url = null*, *$full = false*)

Paramètres

– **$url** (*mixed*) – Une URL relative à Cake, comme "/products/edit/92" ou "/presidents/elect/4" ou un *tableau routing*.
– **$full** (*mixed*) – Si (bool) à true, l'URL entièrement basée sera précédée au résultat. Si un tableau accèpte les clés suivantes.
  – escape - utilisé quand on fait les urls intégrées dans les chaînes de requête html échappées '&'.
  – full - Si à true, l'URL de base complète sera précédée.

Génére une URL pour l'action spécfiée. Retourne une URL pointant vers une combinaison de controller et d'action. $url peut être :
–Empty - la méthode trouve l'adresse du controller/de l'action actuel.
–'/' - la méthode va trouver l'URL de base de l'application.
–Une combinaison de controller/action - la méthode va trouver l'url pour cela.
Il y a quelques paramètres 'spéciaux' qui peuvent changer la chaîne d'URL finale qui est générée :
–`base` - défini à false pour retirer le chemin de base à partir d'URL générée. Si votre application n'est pas le répertoire root, ceci peut être utilisé pour générer les URLs qui sont 'cake relative'. Les URLs Cake relative sont nécessaires quand on utilise requestAction.
–`?` - Prend un tableau de paramètres de chaîne requêté.
–`#` - Vous permet de définir les fragments hashés d'URL.
–`full_base` - Si à true, la valeur de `Router::baseUrl()` sera ajoutée avant aux urls générées.

static `Router::mapResources`(*$controller*, *$options = array()*)

Crée les routes de ressource REST pour les controller(s) donné. Regardez la section *REST* pour plus d'informations.

static `Router::parseExtensions`(*$types*)

Utilisé dans routes.php pour déclarer quelle *Extensions de Fichier* de votre application supporte. En ne fournissant aucun argument, toutes les extensions de fichiers seront supportées. Introduit dans la version 2.1.

static `Router::setExtensions`(*$extensions*, *$merge = true*)

Introduit dans la version 2.2. Défini ou ajoute des extensions valides. Pour avoir des extensions parsées, vous avez toujours besoin d'appeler `Router::parseExtensions()`.

static `Router::defaultRouteClass`(*$classname*)

Définit la route par défaut à utiliser quand on connecte les routes dans le futur.

static `Router::baseUrl`(*$url = null*)

Introduit dans la version 2.4. Récupère ou définit la baseURL utilisée pour la génération d'URLs. Quand vous définissez cette valeur, vous devez vous assurer d'inclure le nom de domaine complètement compétent en incluant le protocole.

Définir les valeurs avec cette méthode va aussi mettre à jour `App.fullBaseUrl` dans `Configure`.

class `CakeRoute`

La classe de base pour les routes personnalisées sur laquelle on se base.

`CakeRoute::parse`(*$url*)

Paramètres

– **$url** (*string*) – La chaîne url à parser.

Parse une url entrante, et génére un tableau de paramètres requêtés sur lequel le Dispatcher peut agir. Etendre cette méthode vous permet de personnaliser comment les URLs entrantes sont converties en un tableau. Retourne `false` à partir d'une URL pour indiquer un échec de match.

`CakeRoute::match`(*$url*)

Paramètres

– **$url** (*array*) – Le tableau de routing pour convertir dans une chaîne URL.

Tente de matcher un tableau URL. Si l'URL matche les paramètres de route et les configurations, alors retourne une chaîne URL générée. Si l'URL ne match pas les paramètres de route, false sera retourné. Cette méthode gère le routing inversé ou la conversion de tableaux d'URL dans des chaînes URLs.

`CakeRoute::compile`()

Forcer une route à compiler son expression régulière.

Sessions

CakePHP fournit des fonctionnalités de wrapper and suite of utility au-dessus de l'extension native `session` de PHP. Les Sessions vous permettent d'identifier les utilisateurs uniques pendant leurs requêtes et de stocker les données persistantes pour les utilisateurs spécifiques. Au contraire des Cookies, les données de session ne sont pas disponibles du coté client. L'utilisation de `$_SESSION` est généralement à éviter dans CakePHP, et à la place l'utilisation des classes de Session est préférable.

Session Configuration

La configuration de Session est stockée dans `Configure`, et les classes de session la récupéreront de là quand c'est nécessaire. La configuration de Session est stockée dans la clé de top niveau `Session`, et un certain nombre d'options sont disponibles :

– `Session.cookie` - Change le nom du cookie de session.
– `Session.timeout` - Le nombre de *minutes* avant que le gestionnaire de session de CakePHP ne fasse expirer la session. Cela affecte `Session.autoRegenerate` (ci-dessous), et cela est geré par CakeSession.

- `Session.cookieTimeout` - Le nombre de *minutes* avant que le cookie de session n'expire. Si il n'est pas défini, il utilisera la même valeur que `Session.timeout`. Cela affecte le cookie de session, et est geré directement par PHP.
- `Session.checkAgent` - Le user agent doit il être verifié, sur chaque requête. Si le useragent ne matche pas, la session sera détruite.
- `Session.autoRegenerate` - Activer cette configuration, allume automatiquement des renouvellements de sessions, et les ids de session qui changent fréquemment. Activer cette valeur va utiliser la valeur `Config.countdown` de la session pour garder une trace des demandes. Une fois que le compte à rebours atteint 0, l'id de session sera regénéré. C'est une bonne option à utiliser pour les applications qui necéssitent de changer fréquemment les ids de session pour des raisons de sécurité. Vous pouvez contrôler le nombre de requêtes necéssaires pour regénérer la session en modifiant `CakeSession::$requestCountdown`.
- `Session.defaults` - Vous permet d'utiliser les configurations de session intégrées par défaut comme une base pour votre configuration de session.
- `Session.handler` - Vous permet de définir un gestionnaire de session personnalisé. La base de données du coeur et les gestionnaires de cache de session utilisent celui-ci. Cette option remplace `Session.save` dans les versions précédentes. Regardez ci-dessous pour des informations supplémentaires sur les gestionnaires de Session.
- `Session.ini` - Vous permet de définir les configurations ini de session supplémentaire pour votre config. Ceci combiné avec `Session.handler` remplace les fonctionnalités de gestionnaire de session personnalisé des versions précédentes.

CakePHP met par défaut la configuration de `session.cookie_secure` à true, quand votre application est sur un protocole SSL. Si votre application sert à partir des deux protocoles SSL et non-SSL, alors vous aurez peut-être des problèmes avec les sessions étant perdues. Si vous avez besoin d'accéder à la session sur les deux domaines SSL et non-SSL, vous aurez envie de désactiver cela :

```
Configure::write('Session', array(
    'defaults' => 'php',
    'ini' => array(
        'session.cookie_secure' => false
    )
));
```

Les chemins des cookies de Session sont par défaut `/` dans 2.0. Pour changer cela, vous pouvez utiliser le drapeau ini `session.cookie_path` vers le chemin du répertoire de votre application :

```
Configure::write('Session', array(
    'defaults' => 'php',
    'ini' => array(
        'session.cookie_path' => '/app/dir'
    )
));
```

Gestionnaires de Session intégrés & configuration

CakePHP est fourni avec plusieurs configurations de session intégrées. Vous pouvez soit utiliser celles-ci comme base pour votre configuration de session, soit vous pouvez créer une solution complètement personnalisée. Pour utiliser les valeurs par défaut, définissez simplement la clé 'defaults' avec le nom par défaut,

vous voulez utiliser. Vous pouvez ensuite surcharger toute sous-configuration en la déclarant dans votre config Session :

```
Configure::write('Session', array(
    'defaults' => 'php'
));
```

Ce qui est au-dessus va utiliser la configuration de session intégrée dans 'php'. Vous pourriez augmenter tout ou partie de celle-ci en faisant ce qui suit :

```
Configure::write('Session', array(
    'defaults' => 'php',
    'cookie' => 'my_app',
    'timeout' => 4320 //3 days
));
```

Ce qui est au-dessus surcharge le timeout et le nom du cookie pour la configuration de session 'php'. Les configurations intégrées sont :

- `php` - Sauvegarde les sessions avec les configurations standard dans votre fichier php.ini.
- `cake` - Sauvegarde les sessions en tant que fichiers à l'intérieur de `app/tmp/sessions`. Ceci est une bonne option quand les hôtes qui ne vous autorisent pas à écrire en dehors de votre propre dir home.
- `database` - Utiliser les sessions de base de données intégrées. Regardez ci-dessous pour plus d'informations.
- `cache` - Utiliser les sessions de cache intégrées. Regardez ci-dessous pour plus d'informations.

Gestionnaires de Session

Les gestionnaires de Session peuvent être aussi définis dans le tableau de config de session. Quand ils sont définis, ils vous permettent de mapper les valeurs divers `session_save_handler` vers une classe ou un objet que vous souhaitez utiliser pour sauvegarder la session. Il y a deux façons d'utiliser le 'handler'. La première est de fournir un tableau avec 5 callables. Ces callables sont ensuite appliqués à `session_set_save_handler` :

```
Configure::write('Session', array(
    'userAgent' => false,
    'cookie' => 'my_cookie',
    'timeout' => 600,
    'handler' => array(
        array('Foo', 'open'),
        array('Foo', 'close'),
        array('Foo', 'read'),
        array('Foo', 'write'),
        array('Foo', 'destroy'),
        array('Foo', 'gc'),
    ),
    'ini' => array(
        'cookie_secure' => 1,
        'use_trans_sid' => 0
    )
));
```

Le deuxième mode est de définir une clé 'engine'. Cette clé devrait être un nom de classe qui implémente `CakeSessionHandlerInterface`. Implémenter cette interface va autoriser CakeSession à mapper automatiquement les méthodes pour le gestionnaire. Les deux gestionnaires de Session du Cache du Coeur et de la base de données utilisent cette méthode pour sauvegarder les sessions. Les configurations supplémentaires pour le gestionnaire doivent être placées à l'intérieur du tableau handler. Vous pouvez ensuite lire ces valeurs à partir de l'intérieur de votre handler.

Vous pouvez aussi utiliser les gestionnaires de session à partir des plugins. En configurant le moteur avec quelque chose comme `MyPlugin.PluginSessionHandler`. Cela ca charger et utiliser la classe `PluginSessionHandler` à partir de l'intérieur du MyPlugin de votre application.

CakeSessionHandlerInterface

Cette interface est utilisée pour tous les gestionnaires de session personnalisés à l'intérieur de CakePHP, et peut être utilisé pour créer des gestionnaires de session personnalisées user land. En implémentant simplement l'interface dans votre classe et en définissant `Session.handler.engine` au nom de classe que vous avez crée. CakePHP va tenter de charger le gestionnaire à partir de l'intérieur de `app/Model/Datasource/Session/$classname.php`. Donc si votre nom de classe est `AppSessionHandler`, le fichier devrait être `app/Model/Datasource/Session/AppSessionHandler.php`.

Les sessions de la Base de Données

Les changements dans la configuration de session changent la façon dont vous définissez les sessions de base de données. La plupart du temps, vous aurez seulement besoin de définir `Session.handler.model` dans votre configuration ainsi que choisir la base de données par défaut :

```
Configure::write('Session', array(
    'defaults' => 'database',
    'handler' => array(
        'model' => 'CustomSession'
    )
));
```

Ce qui est au-dessus va dire à CakeSession d'utiliser le 'database' intégré par défaut, et spécifier qu'un model appelé `CustomSession` sera celui délégué pour la sauvegarde d'information de session dans la base de données.

Les Sessions de Cache

La classe Cache peut aussi être utilisée pour stocker les sessions. Cela vous permet de stocker les sessions dans un cache comme APC, memcache, ou Xcache. Il y a some caveats dans l'utilisation des sessions en cache, puisque si vous exhaust l'espace de cache, les sessions vont commencer à expirer puisque les enregistrements sont évincés.

Pour utiliser les sessions basées sur le Cache, vous pouvez configurer votre config Session comme ceci

```
Configure::write('Session', array(
    'defaults' => 'cache',
    'handler' => array(
        'config' => 'session'
    )
));
```

Cela va configurer CakeSession pour utiliser la classe `CacheSession` déléguée pour sauvegarder les sessions. Vous pouvez utiliser la clé 'config' qui va mettre en cache la configuration à utiliser. La configuration par défaut de la mise en cache est `'default'`.

Configurer les directives ini

Celui intégré par défaut tente de fournir une base commune pour la configuration de session. Vous aurez aussi besoin d'ajuster les flags ini spécifiques. CakePHP donne la capacité de personnaliser les configurations ini pour les deux configurations par défaut, ainsi que celles personnalisées. La clé `ini` dans les configurations de session vous permet de spécifier les valeurs de configuration individuelles. Par exemple vous pouvez l'utiliser pour contrôler les configurations comme `session.gc_divisor` :

```
Configure::write('Session', array(
    'defaults' => 'php',
    'ini' => array(
        'session.gc_divisor' => 1000,
        'session.cookie_httponly' => true
    )
));
```

Créer un gestionnaire de session personnalisé

Créer un gestionnaire de session personnalisé est simple dans CakePHP. Dans cet exemple, nous allons créer un gestionnaire de session qui stocke les sessions à la fois dans le Cache (apc) et la base de données. Cela nous donne le meilleur du IO rapide de apc, sans avoir à se soucier des sessions s'évaporant quand le cache se remplit.

D'abord, nous aurons besoin de créer notre classe personnalisée et de la mettre dans `app/Model/Datasource/Session/ComboSession.php`. La classe devrait ressembler à :

```
App::uses('DatabaseSession', 'Model/Datasource/Session');

class ComboSession extends DatabaseSession implements CakeSessionHandlerInterface {
    public $cacheKey;

    public function __construct() {
        $this->cacheKey = Configure::read('Session.handler.cache');
        parent::__construct();
    }

    // Lit les données à partir d'une session.
    public function read($id) {
```

```
        $result = Cache::read($id, $this->cacheKey);
        if ($result) {
            return $result;
        }
        return parent::read($id);
    }

    // écrit les données dans la session.
    public function write($id, $data) {
        $result = Cache::write($id, $data, $this->cacheKey);
        if ($result) {
            return parent::write($id, $data);
        }
        return false;
    }

    // détruit une session.
    public function destroy($id) {
        $result = Cache::delete($id, $this->cacheKey);
        if ($result) {
            return parent::destroy($id);
        }
        return false;
    }

    // retire les sessions expirées.
    public function gc($expires = null) {
        return Cache::gc($this->cacheKey) && parent::gc($expires);
    }
}
```

Notre classe étend la classe intégrée `DatabaseSession` donc nous ne devons pas dupliquer toute sa logique et son comportement. Nous entourons chaque opération avec une opération `Cache`. Cela nous laisse récupérer les sessions de la mise en cache rapide, et nous évite de nous inquiéter sur ce qui arrive quand nous remplissons le cache. Utiliser le gestionnaire de session est aussi facile. Dans votre `core.php` imitez le block de session ressemblant à ce qui suit :

```
Configure::write('Session', array(
    'defaults' => 'database',
    'handler' => array(
        'engine' => 'ComboSession',
        'model' => 'Session',
        'cache' => 'apc'
    )
));

// Assurez vous d'ajouter une config de cache apc
Cache::config('apc', array('Engine' => 'Apc'));
```

Maintenant notre application va commencer en utilisant notre gestionnaire de session personnalisé pour la lecture & l'écriture des données de session.

class `CakeSession`

Lire & écrire les données de session

Selon le contexte dans lequel vous êtes dans votre application, vous avez différentes classes qui fournissent un accès à la session. Dans les controllers, vous pouvez utiliser `SessionComponent`. Dans la vue, vous pouvez utiliser `SessionHelper`. Dans tout autre partie de votre application, vous pouvez utiliser `CakeSession` pour accéder aussi à la session. Comme les autres interfaces de session, `CakeSession` fournit une interface simple de CRUD.

static `CakeSession::read`(*$key*)

Vous pouvez lire les valeurs de session en utilisant la syntaxe compatible `Set::classicExtract()` :

```
CakeSession::read('Config.language');
```

static `CakeSession::write`(*$key*, *$value*)

`$key` devrait être le chemin séparé de point et `$value` sa valeur :

```
CakeSession::write('Config.language', 'eng');
```

static `CakeSession::delete`(*$key*)

Quand vous avez besoin de supprimer des données à partir de la session, vous pouvez utiliser delete :

```
CakeSession::delete('Config.language');
```

Vous devriez aussi voir la documentation sur *Sessions* et *SessionHelper* sur la façon d'accéder aux données de Session dans le controller et la vue.

Exceptions

Les Exceptions peuvent être utilisées pour une variété d'utilisations dans votre application. CakePHP utilise les exceptions en interne pour indiquer les erreurs logiques ou les erreurs d'utilisation. Toutes les exceptions levées de CakePHP étendent `CakeException`, et il y a des exceptions spécifiques selon les classes/tâches qui étendent la classe de base.

CakePHP fournit aussi un nombre de classes d'exceptions qui peuvent être utilisées pour les erreurs HTTP. Regardez la section sur *Exceptions intégrées pour CakePHP* pour plus d'informations.

Configuration de Exception

Il y a certaines clés disponibles pour configurer les exceptions :

```
Configure::write('Exception', array(
    'handler' => 'ErrorHandler::handleException',
    'renderer' => 'ExceptionRenderer',
    'log' => true
));
```

– `handler` - callback - Le callback pour gérer les exceptions. Vous pouvez définir ceci pour n'importe quel type de callback, incluant les fonctions anonymes.

- `renderer` - string - La classe responsable du rendu des exceptions non attrapées. Si vous choisissez une classe personnalisée, vous devriez placer ce fichier pour cette classe dans app/Lib/Error. Cette classe a besoin d'implémenter une méthode `render()`.
- `log` - boolean - Quand à true, les exceptions + leurs stack traces seront logged à CakeLog.
- `consoleHandler` - callback - The callback used to handle exceptions, in a console context. If undefined, CakePHP's default handler will be used.

Le rendu d'Exception par défaut affiche une page HTML, vous pouvez personnaliser soit le gestionnaire soit le rendu en changeant les configurations. Changer le gestionnaire, vous permet de prendre le contrôle total sur le processus de gestion d'exception, tandis que changer le rendu vous permet de changer facilement la sortie type/contenu, ainsi que d'ajouter une gestion d'exception spécifique dans l'application. Introduit dans la version 2.2 : L'option `Exception.consoleHandler` a été ajoutée dans 2.2.

Classes d'Exception

Il y a un certain nombre de classes d'exception dans CakePHP. Chaque exception remplace un message d'erreur `cakeError()` du passé. Les Exceptions offrent une flexibilité supplémentaire dans laquelle elles peuvent étendre et contenir de la logique. L'exception intégrée va capturer toute exception non attrapée et rendre une page utile. Les Exceptions qui n'utilisent pas spécifiquement un code 400, seront traitées comme une Erreur Interne du Serveur.

Exceptions intégrées pour CakePHP

Il y a plusieurs exceptions intégrées dans CakePHP, en-dehors des exceptions internes du framework, il y a plusieurs exceptions pour les méthodes HTTP.

exception `BadRequestException`
Utilisé pour faire une erreur 400 de Mauvaise Requête.

exception `ForbiddenException`
Utilisé pour faire une erreur 403 Interdite.

exception `NotFoundException`
Utilisé pour faire une erreur 404 Non Trouvé.

exception `MethodNotAllowedException`
Utilisé pour faire une erreur 405 pour les Méthodes Non Autorisées.

exception `InternalErrorException`
Utilisé pour faire une Erreur 500 du Serveur Interne.

exception `NotImplementedException`
Utilisé pour faire une Erreur 501 Non Implémentée.

Vous pouvez lancer ces exceptions à partir de vos controllers pour indiquer les états d'échec, ou les erreurs HTTP. Un exemple d'utilisation des exceptions HTTP pourraient rendre les pages 404 pour les items qui n'ont pas été trouvés

Vous pouvez lancer ces exceptions à partir de vos controllers pour indiquer les états d'échecs, ou les erreurs HTTP. Un exemple d'utilisation des exceptions HTTP pourrait être le rendu de pages 404 pour les items qui n'ont pas été trouvés :

```
public function view ($id) {
    $post = $this->Post->findById($id);
    if (!$post) {
        throw new NotFoundException('N a pas trouvé ce post');
    }
    $this->set('post', $post);
}
```

En utilisant les exceptions pour les erreurs HTTP, vous pouvez garder à la fois votre code propre, et donner les réponses complètement REST aux appications clientes et aux utilisateurs.

De plus, les exceptions de couche du framework suivantes sont disponibles, et seront lancées à partir de certains components du coeur de CakePHP :

exception `MissingViewException`
Le fichier de vue choisi n'a pas pu être trouvé.

exception `MissingLayoutException`
Le layout choisi n'a pas pu être trouvé.

exception `MissingHelperException`
Un helper n'a pas pu être trouvé.

exception `MissingBehaviorException`
Un behavior configuré n'a pas pu être trouvé.

exception `MissingComponentException`
Un component configuré n'a pas pu être trouvé.

exception `MissingTaskException`
Une tâche configurée n'a pas pu être trouvée.

exception `MissingShellException`
La classe shell n'a pas pu être trouvée.

exception `MissingShellMethodException`
La classe de shell choisi n'a pas de méthode avec ce nom.

exception `MissingDatabaseException`
La base de donnée configurée n'existe pas.

exception `MissingConnectionException`
Une connection à un model n'existe pas.

exception `MissingTableException`
Une table de model est manquante.

exception `MissingActionException`
L'action du controller requêté n'a pas pu être trouvé.

exception `MissingControllerException`
Le controller requêté n'a pas pu être trouvé.

exception `PrivateActionException`
Accès privé à l'action. Soit les actions ont un accès privé/protegé/préfixé par _, ou essaient d'accéder

aux routes préfixés de manière incorrecte.

exception `CakeException`
Classe d'exception de base dans CakePHP. Toutes les exceptions lancées par CakePHP étendront cette classe.

Ces classes d'exception étendent toutes `CakeException`. En étendant CakeException, vous pouvez créer vos propres erreurs 'framework'. Toutes les Exceptions standardes que CakePHP va aussi lancer les CakeException étendues. Introduit dans la version 2.3 : CakeBaseException a été ajoutée

exception `CakeBaseException`
La classe d'exception de base dans CakePHP. Toutes les CakeExceptions et HttpExceptions ci-dessus étendent cette classe.

`CakeBaseException::responseHeader`(*$header = null*, *$value = null*)
Voir `CakeResponse::header()`.

Toutes les exceptions Http et Cake étendent la classe CakeBaseException, qui a une méthode pour ajouter les en-têtes à la réponse. Par exemple quand vous lancez une MethodNotAllowedException 405, le rfc2616 dit : "La réponse DOIT inclure un en-tête contenant une liste de méthodes valides pour la ressource requêtée."

Utiliser les exceptions HTTP dans vos controllers

Vous pouvez envoyer n'importe quelle exception HTTP liée à partir des actions de votre controller pour indiquer les états d'échec. Par exemple :

```
public function view($id) {
    $post = $this->Post->read(null, $id);
    if (!$post) {
        throw new NotFoundException();
    }
    $this->set(compact('post'));
}
```

Ce qui est au-dessus causerait l'`Exception.handler` configurée pour attraper et traiter `NotFoundException`. Par défaut, cela va créer une page d'erreur et enregistrer l'exception. Exception Renderer ==========s

class `ExceptionRenderer`(*Exception $exception*)

La classe ExceptionRenderer avec l'aide de `CakeErrorController` s'occupe du rendu des pages d'erreur pour toutes les exceptions lancées par votre application.

Les vues de la page d'erreur sont localisées dans `app/View/Errors/`. Pour toutes les erreurs 4xx et 5xx, les fichiers de vue `error400.ctp` et `error500.ctp` sont utilisées respectivement. Vous pouvez les personnaliser selon vos besoins. Par défaut, votre `app/Layouts/default.ctp` est utilisé aussi pour les pages d'erreur. Si par exemple, vous voulez utiliser un autre layout `app/Layouts/my_error.ctp` pour vos pages d'erreur, alors modifiez simplement les vues d'erreur et ajoutez le statement `$this->layout = 'my_error';` à `error400.ctp` et `error500.ctp`.

Chaque exception de layer framework a son propre fichier de vue localisé dans les templates du coeur mais vous n'avez pas besoin de personnaliser les deux puisqu'ils sont utilisés seulement pen-

dant le développement. Avec debug éteint, toutes les exceptions du layer framework sont converties en `InternalErrorException`.

Créer vos propres exceptions dans votre application

Vous pouvez créer vos propres exceptions d'application en utilisant toute exception SPL (http ://php.net/manual/en/spl.exceptions.php) intégrée, `Exception` lui-même, ou `CakeException`. Les exceptions d'Application qui étendent les Exceptions ou les exceptions SPL vont être traitées comme une erreur 500 dans le mode de production. `CakeException` est spécial dans le fait que tous les objets `CakeException` sont contraints d'être soit dans des erreurs 500 soit 404, selon le code qu'ils utilisent. Quand vous êtes en mode développement, les objets `CakeException` ont besoin simplement d'un nouveau template qui matche le nom de classe afin fournir des informations utiles. Si votre application contenait l'exception suivante :

```
class MissingWidgetException extends CakeException {};
```

Vous pourriez fournir des erreurs de bon développement, en créant `app/View/Errors/missing_widget.ctp`. Quand on est en mode production, l'erreur du dessus serait traitée comme une erreur 500. Le constructeur pour `CakeException` a été étendu, vous autorisant à passer des données hashées. Ces hashs sont interpolés dans le messageTemplate, ainsi que dans la vue qui est utilisée pour représenter l'erreur dans le mode développement. Cela vous permet de créer des exceptions de données riches, en fournissant plus de contexte pour vos erreurs. Vous pouvez aussi fournir un template de message qui permet les méthodes natives `__toString()` pour fonctionner normalement :

```
class MissingWidgetException extends CakeException {
    protected $_messageTemplate = 'Seems that %s is missing.';
}

throw new MissingWidgetException(array('widget' => 'Pointy'));
```

Quand attrapé par le gestionnaire d'exception intégré, vous obtiendriez une variable `$widget` dans votre template de vue d'erreur. De plus, si vous attrapez l'exception en chaîne ou utilisez sa méthode `getMessage()`, vous auriez `Il semble que Pointy soit manquant.`. Cela vous permet de créer facilement et rapidement vos propres erreurs de développement riche, juste comme CakePHP en interne.

Créer des codes de statut personnalisés

Vous pouvez créer des codes de statut HTTP personnalisés en changeant le code utilisé quand vous créez une exception :

```
throw new MissingWidgetHelperException('Its not here', 501);
```

Va créer un code de réponse `501`, vous pouvez utiliser le code de statut HTTP que vous souhaitez. En développement, si votre exception n'a pas de template spécifique, et que vous utilisez un code égal ou supérieur à `500`, vous verrez le template `error500`. Pour toute autre code d'erreur, vous aurez le template `error400`. Si vous avez défini un template d'erreur pour votre exception personnalisée, ce template va

être utilisé en mode développement. Si vous souhaitez votre propre gestionnaire d'exception logique même en production, regardez la section suivante.

Etendre et Implementer vos propres gestionnaires d'Exception

Vous pouvez implémenter un gestionnaire d'exception spécifique pour votre application de plusieurs façons. Chaque approche vous donne différents montants de contrôle sur le processus de gestion d'exception.
- Set `Configure::write('Exception.handler', 'YourClass::yourMethod');`
- Create `AppController::appError();`
- Set `Configure::write('Exception.renderer', 'YourClass');`

Dans les prochaines sections, nous allons détailler les différentes approches et les bénéfices de chacun.

Créer vos propress gestionnaires d'Exception avec *Exception.handler*

Créer votre propre gestionnaire d'exception vous donne plus de contrôle sur le processus de gestion des exceptions. La classe que vous choisissez devra être chargée dans votre `app/Config/bootstrap.php`, ainsi elle sera disponible pour gérer toute exception. Vous pouvez définir le gestionnaire comme tout type de callback. En configurant `Exception.handler` CakePHP va ignorer toutes les configurations d'Exception. Une configuration de gestionnaire d'exception personnalisée pourrait par exemple ressembler à ceci :

```
// dans app/Config/core.php
Configure::write('Exception.handler', 'AppExceptionHandler::handle');

// dans app/Config/bootstrap.php
App::uses('AppExceptionHandler', 'Lib');

// dans app/Lib/AppExceptionHandler.php
class AppExceptionHandler {
    public static function handle($error) {
        echo 'Oh noes! ' . $error->getMessage();
        // ...
    }
    // ...
}
```

Vous pouvez lancer tout code que vous souhaitez à l'intérieur de `handleException`. Le code ci-dessus afficherait simplement 'Oh noes ! ' plus le message d'exception. Vouspouvez définir des gestionnaires d'exception comme tout type de callback, même une fonction anonyme si vous utilisez PHP 5.3 :

```
Configure::write('Exception.handler', function ($error) {
    echo 'Ruh roh ' . $error->getMessage();
});
```

En créant un gestionnaire d'exception personnalisé, vous pouvez fournir un gestionnaire d'erreur personnalisé pour les exceptions de l'application. Dans la méthode fournie comme un gestionnaire d'exception, vous pourriez faire comme suit :

```
// dans app/Lib/AppErrorHandler.php
class AppErrorHandler {
    public static function handleException($error) {
        if ($error instanceof MissingWidgetException) {
            return self::handleMissingWidget($error);
        }
        // faire d'autres trucs.
    }
}
```

Utiliser AppController : :appError() ;

Implémenter cette méthode est une alternative pour implémenter un gestionnaire d'exception personnalisé. Il est fourni principalement pour une compatibilité backwards, et il n'est pas recommandé pour les nouvelles applications. Cette méthode de controller est appelée à la place du rendu d'exception par défaut. Il reçoit l'exception lancée comme son seul argument. Vous devriez implémenter votre gestionnaire d'erreur dans cette méthode :

```
class AppController extends Controller {
    public function appError($error) {
        // logique personnalisée va ici.
    }
}
```

Utiliser un rendu personnalisé avec Exception.renderer pour gérer les exceptions d'application

Si vous ne voulez pas prendre contrôle du gestionnaire d'exception, mais que vous voulez changer la façon dont les exceptions sont rendues, vous pouvez utiliser `Configure::write('Exception.renderer','AppExceptionRenderer');` pour choisir une classe qui va rendre les pages d'exception. Par défaut :php :class‘ExceptionRenderer‘ est utilisée. Votre classe de rendu d'exception personnalisée doit être placée dans `app/Lib/Error`. Ou un répertoire `Error` ‘ dans tout chemin bootstrapped Lib. Dans une classe de rendu d'exception, vous pouvez fournir une gestion spécialisée pour les erreurs spécifiques de l'application :

```
// dans app/Lib/Error/AppExceptionRenderer.php
App::uses('ExceptionRenderer', 'Error');

class AppExceptionRenderer extends ExceptionRenderer {
    public function missingWidget($error) {
        echo 'Oops that widget is missing!';
    }
}
```

Ce qui est au-dessus gérerait tout exception de type `MissingWidgetException`, et vous permettrait de fournir une logique d'affichage/de gestionnaire personnalisé pour ces applications. Les méthodes de gestion d'exception récupèrent l'exception en étant géré comme leur argument.

Note : Votre rendu personnalisé devrait avoir une exception comme constructeur, et implémenter une méthode de rendu. Ne pas le faire entraînera des erreurs supplémentaires.

Note : Si vous utilisez un `Exception.handler` personnalisé, cette configuration n'aura aucun effet. A moins que vous le référenciez à l'intérieur de votre implémentation.

Créer un controller personnalisé pour gérer les exceptions

Dans votre sous-classe ExceptionRenderer, vous pouvez utiliser la méthode `_getController` pour vous permettre de retourner un controller personnalisé pour gérer vos erreurs/ Par défaut, CakePHP utilise `CakeErrorController` qui enlève quelques callbacks habituels pour aider à s'assurer que les erreurs s'affichent toujours. Cependant, vous aurez peut-être besoin d'un controller de gestionnaire d'erreur plus personnalisé dans votre application. En implémentant `_getController` dans votre classe `AppExceptionRenderer`, vous pouvez utiliser tout controller que vous souhaitez :

```
class AppExceptionRenderer extends ExceptionRenderer {
    protected function _getController($exception) {
        App::uses('SuperCustomError', 'Controller');
        return new SuperCustomErrorController();
    }
}
```

De façon alternative, vous pouvez simplement écraser le CakeErrorController du coeur, en en incluant un dans `app/Controller`. Si vous utilisez un controller personnalisé pour la gestion des erreurs, assurez-vous de faire toutes les configurations dont vous aurez besoin dans votre constructeur, ou dans la méthode de rendu. Puisque celles-ci sont les seules méthodes que la classe `ErrorHandler` intégrée appelle directement.

Logging exceptions

Utiliser la gestion d'exception intégrée, vous pouvez lancer les exceptions qui sont gérées avec ErrorHandler en configurant `Exception.log` à true dans votre core.php. Activer cela va lacer chaque exception vers `CakeLog` et les loggers configurés.

Note : Si vous utilisez un `Exception.handler` personnalisé, cette configuration n'aura aucun effet. A moins que vous le référenciez à l'intérieur de votre implémentation.

Gestion des Erreurs

Pour 2.0 `Object::cakeError()` a été retirée. A la place, elle a été remplacé par un certain nombre d'exceptions. Toutes les classes du coeur qui appelaient avant cakeError envoient maintenant des exceptions. Cela vous laisse choisir soit la gestion des erreurs dans le code de votre application, soit laisser la gestion intégrée des exceptions le faire pour vous.

Il y a plus de contrôle que jamais pour les erreurs et la gestion des exceptions dans CakePHP 2.0. Vous pouvez configurer quelles méthodes vous voulez définir en tant que gestionnaire d'erreur, et en gestionnaire d'exception en utilisant configure.

Configuration des Erreurs

La configuration des Erreurs est faite à l'intérieur du fichier `app/Config/core.php` de votre application. Vous pouvez définir un callback pour qu'il soit effectué à chaque fois que votre application attrape une erreur PHP - les exceptions sont gérées séparément *Exceptions*. Le callback peut être n'importe quel PHP appelable, avec la possibilité d'appeler une fonction anonyme. L'erreur par défaut de la configuration de gestion ressemble à ceci :

```
Configure::write('Error', array(
    'handler' => 'ErrorHandler::handleError',
    'level' => E_ALL & ~E_DEPRECATED,
    'trace' => true
));
```

Vous avez 5 options intégrées quand vous gérez la configuration des erreurs :

- `handler` - callback - Le callback pour la gestion des erreurs. Vous pouvez définir ceci à n'importe quel type, incluant des fonctions anonymes.
- `level` - int - Le niveau d'erreurs qui vous interesse dans la capture. Utilisez les constantes d'erreur intégrées dans PHP, et bitmasks pour séléctionner le niveau d'erreur qui vous intéressent.
- `trace` - boolean - Inclut stack traces pour les erreurs dans les fichiers de log. Les Stack traces seront inclus dans le log après chaque erreur. C'est utile pour trouver où/quand les erreurs ont été faites.
- `consoleHandler` - callback - Le callback utilisé pour gérer les erreurs quand vous lancez la console. Si il n'est pas défini, les gestionnaires par défaut de CakePHP seront utilisés.

ErrorHandler par défaut, affiche les erreurs quand `debug` > 0, et les erreurs de logs quand debug = 0. Le type d'erreurs capté dans les deux cas est contrôlé par `Error.level`. Le gestionnaire d'erreurs fatales va être appelé indépendamment du niveau de `debug` ou de la configuration de `Error.level`, mais le résultat va être différent, basé sur le niveau de `debug`.

Note : Si vous utilisez un gestionnaire d'erreur personnalisé, le trace setting n'aura aucun effet, à moins que vous y fassiez référence dans votre fonction de gestion d'erreur.

Introduit dans la version 2.2 : L'option `Error.consoleHandler` a été ajoutée dans 2.2.Modifié dans la version 2.2 : Les `Error.handler` et `Error.consoleHandler` vont recevoir aussi les codes d'erreur fatal. Le comportement par défaut est de montrer une page d'erreur interne du serveur (`debug` désactivé) ou une page avec le message, fichier et la ligne (`debug` activé).

Créer vos propres gestionnaires d'erreurs

Vous pouvez créer un gestionnaire d'erreur à partir de n'importe quel type de callback. Par exemple, vous pouvez utiliser une classe appelée `AppError` pour gérer vos erreurs. Ce qui suit serait à faire :

```
//dans app/Config/core.php
Configure::write('Error.handler', 'AppError::handleError');

//dans app/Config/bootstrap.php
App::uses('AppError', 'Lib');

//dans app/Lib/AppError.php
class AppError {
    public static function handleError($code, $description, $file = null, $line = null, $co
        echo 'Il y a eu une erreur!';
    }
}
```

Cette classe/méthode va afficher 'Il y a eu une erreur !' chaque fois qu'une erreur apparaît. Depuis que vous pouvez définir un gestionnaire d'erreur comme tout type de callback, vous pouvez utiliser une fonction anonyme si vous utilisez PHP5.3 ou supérieur. :

```
Configure::write('Error.handler', function($code, $description, $file = null, $line = null,
    echo 'Oh non quelque chose est apparu';
});
```

Il est important de se rappeler que les erreurs captées par le gestionnaire d'erreurs configuré seront des erreurs php, et si vous avez besoin de gestion d'erreurs personnalisée, vous aurez probablement aussi envie de configurer la gestion des *Exceptions*.

Changer le comportement des erreurs fatales

Depuis CakePHP 2.2, `Error.handler` va aussi recevoir les codes d'erreur fatal. Si vous ne voulez pas montrer la page d'erreur de cake, vous pouvez la remplacer comme cela :

```
//dans app/Config/core.php
Configure::write('Error.handler', 'AppError::handleError');

//dans app/Config/bootstrap.php
App::uses('AppError', 'Lib');

//dans app/Lib/AppError.php
class AppError {
    public static function handleError($code, $description, $file = null, $line = null, $co
        list(, $level) = ErrorHandler::mapErrorCode($code);
        if ($level === LOG_ERROR) {
            // Ignore l\'erreur fatale. Cela ne va garder seulement le message d\'erreur PH
            return false;
        }
        return ErrorHandler::handleError($code, $description, $file, $line, $context);
    }
}
```

Si vous voulez garder le comportement d'erreur fatal par défaut, vous pouvez appeler `ErrorHandler::handleFatalError()` à partir du gestionnaire personnalisé.

Debugger

Le debug est une inévitable et nécessaire partie de tout cycle de développement. Tandis que CakePHP n'offre pas d'outils qui se connectent directement avec tout IDE ou éditeur, CakePHP fournit plusieurs outils pour l'aide au debug et ce qui est lancé sous le capot de votre application.

Debug basique

debug (*mixed $var*, *boolean $showHtml = null*, *$showFrom = true*)

Paramètres

- **$var** (*mixed*) – Les contenus à afficher. Les tableaux et objets fonctionnent bien.
- **$showHTML** (*boolean*) – Défini à true, pour activer l'échappement. L'échappement est activé par défaut dans 2.0 quand on sert les requêtes web.
- **$showFrom** (*boolean*) – Montre la ligne et le fichier pour lesquels le debug() apparaît.

La fonction debug() est une fonction disponible partout qui fontionne de la même manière que la fonction PHP print_r(). La fonction debug() vous permet de montrer les contenus d'un variable de différentes façons. Premièrement, si vous voulez que vos données soient montrées d'une façon sympa en HTML, définissez le deuxième paramètre à true. La fonction affiche aussi la ligne et le fichier dont ils sont originaires par défaut.

La sortie de cette fonction est seulement montrée si la variable de debug du coeur a été définie à une valeur supérieure à 0. Modifié dans la version 2.1 : La sortie de `debug()` ressemble plus `var_dump()`, et utilise `Debugger` en interne.

Classe Debugger

La classe debugger a été introduite avec CakePHP 1.2 et offre même plus d'options pour obtenir les informations de debug. Elle a plusieurs fonctions qui sont appelées statiquement, et fournissent le dumping, logging et les fonctions de gestion des erreurs.

La Classe Debugger écrase la gestion des erreurs PHP par défaut, le remplaçant avec bien plus de rapports d'erreurs utiles. La gestion des erreurs de Debugger est utilisée par défaut dans CakePHP. Comme pour toutes les fonctions de debug, `Configure::debug` doit être défini à une valeur supérieure à 0.

Quand une erreur est levée, Debugger affiche à la fois l'information de la page et fait une entrée dans le fichier error.log. Le rapport d'erreurs qui est généré a les deux stack trace et un extrait de où l'erreur a été levée. Cliquez sur le type de lien "Error" pour revéler le stack trace, et sur le lien "Code" pour revéler les lignes d'erreurs en cause.

Utiliser la Classe Debugger

class Debugger

Pour utiliser le debugger, assurez-vous d'abord que Configure : :read('debug') est défini à une valeur supérieure à 0.

static `Debugger::dump($var)`

Dump prints out the contents of a variable. Elle affiche toutes les propriétés et méthodes (si il y en a) de la variable fournie :

```
$foo = array(1,2,3);

Debugger::dump($foo);

// sortie
array(
    1,
    2,
    3
)

// objet simple
$car = new Car();

Debugger::dump($car);

// sortie
Car
Car::colour = 'red'
Car::make = 'Toyota'
Car::model = 'Camry'
Car::mileage = '15000'
Car::acclerate()
Car::decelerate()
Car::stop()
```

Modifié dans la version 2.1 : Dans 2.1 forward, la sortie a été modifiée pour la lisibilité. Regardez `Debugger::exportVar()`.

static `Debugger::log($var, $level = 7)`

Crée un stack trace log détaillé au moment de l'invocation. La méthode log() affiche les données identiques à celles faites par Debugger : :dump(), mais dans debug.log au lieu de les sortir buffer. Notez que votre répertoire app/tmp directory (et son contenu) doit être ouvert en écriture par le serveur web pour que le log() fonctionne correctement.

static `Debugger::trace($options)`

Retourne le stack trace courant. Chaque ligne des traces inlut la méthode appelée, incluant chaque fichier et ligne d'où est originaire l'appel. :

```
//Dans PostsController::index()
pr( Debugger::trace() );

//sorties
PostsController::index() - APP/Controller/DownloadsController.php, line 48
Dispatcher::_invoke() - CORE/lib/Cake/Routing/Dispatcher.php, line 265
Dispatcher::dispatch() - CORE/lib/Cake/Routing/Dispatcher.php, line 237
[main] - APP/webroot/index.php, line 84
```

Ci-dessus se trouve le stack trace généré en appelant Debugger : :trace() dans une action d'un controller. Lire le stack trace de bas en haut montre l'ordre des fonctions lancées actuellement (stack

frames). Dans l'exemple du dessus, index.php appelé Dispatcher : :dispatch(), qui est appelé in-turn Dispatcher : :_invoke(). La méthode _invoke() appelé ensuite par PostsController : :index(). Cette information est utile quand vous travaillez avec des opérations récursives ou des stacks profonds, puisqu'il identifie les fonctions qui sont actuellement lancées au moment du trace().

static `Debugger::excerpt` (*$file*, *$line*, *$context*)
Récupérer un extrait du fichier dans $path (qui est un chemin de fichier absolu), mettant en évidence le numéro de la ligne $line avec le nombre de lignes $context autour. :

```
pr( Debugger::excerpt(ROOT.DS.LIBS.'debugger.php', 321, 2) );

//sortira ce qui suit.
Array
(
    [0] => <code><span style="color: #000000"> * @access public</span></code>
    [1] => <code><span style="color: #000000"> */</span></code>
    [2] => <code><span style="color: #000000">    function excerpt($file, $line, $cont

    [3] => <span class="code-highlight"><code><span style="color: #000000">        $da
    [4] => <code><span style="color: #000000">        $data = @explode("\n", file_get_
)
```

Bien que cette méthode est utilisée en interne, elle peut être pratique si vous créez vos propres messages d'erreurs ou les logs pour les situations personnalisées.

static `Debugger::exportVar` (*$var*, *$recursion = 0*)
Convertir une variable de tout type en une chaîne de caractères pour l'utilisation dans la sortie de debug. Cette méthode est aussi utilisée par la plupart de Debugger pour les conversions de variable en interne, et peut aussi être utilisée dans vos propres Debuggers. Modifié dans la version 2.1 : Cette fonction génère une sortie différente dans 2.1 et suivants.

static `Debugger::invoke` (*$debugger*)
Remplace le Debugger de CakePHP avec une nouvelle instance.

static `Debugger::getType` (*$var*)
Récupère le type de variable. Les objets retourneront leur nom de classe. Introduit dans la version 2.1.

Utiliser Logging pour debug

Les messages de Logging est une autre bonne façon de debugger les applications, et vous pouvez utiliser `CakeLog` pour faire le logging dans votre application. Tous les objets qui étendent `Object` ont une méthode d'instanciation *log()* qui peut être utilisé pour les messages de log :

```
$this->log('Got here', 'debug');
```

Ce qui est au-dessus écrit `Got here` dans le debug du log. Vous pouvez utiliser les logs (log entries) pour aider les méthodes de débug qui impliquent les redirections ou les boucles compliquées. Vous pouvez aussi utiliser `CakeLog::write()` pour écrire les messages de log. Cette méthode peut être appelée statiquement partout dans votre application où CakeLog a été chargée :

```
// dans app/Config/bootstrap.php
App::uses('CakeLog', 'Log');

// N'importe où dans votre application
CakeLog::write('debug', 'Got here');
```

Kit de Debug

DebugKit est un plugin qui fournit un nombre de bons outiles de debug. Il fournit principalement une barre d'outils dans le HTML rendu, qui fournit une pléthore d'informations sur votre application et la requête courante. Vous pouvez télécharger DebugKit (https ://github.com/cakephp/debug_kit/tree/2.0) sur github.

Testing

CakePHP est fourni avec un support de test intégré compréhensible. CakePHP permet l'intégration de PHPUnit (http ://phpunit.de). En plus de toutes les fonctionnalités offertes par PHPUnit, CakePHP offre quelques fonctionnalités supplémentaires pour faciliter le test. Cette section va couvrir l'installation de PHPUnit, comment commencer avec le Test Unitaire, et comment vous pouvez utiliser les extensions que CakePHP offre.

Installer PHPUnit

CakePHP utilise PHPUnit as its underlying test framework. PHPUnit est le standard de-facto pour le test unitaire dans PHP. Il offre un ensemble de fonctionnalités profondes et puissantes pour s'assurer que votre code fait ce que vous pensez qu'il doit faire. PHPUnit peut être installé à travers le pear installer (http ://pear.php.net). Pour installer PHPUnit, lancez ce qui suit :

```
pear upgrade PEAR
pear config-set auto_discover 1
pear install pear.phpunit.de/PHPUnit
```

Note : Selon la configuration de votre système, vous devrez lancer les commandes précédentes avec `sudo`.

Une fois que PHPUnit est installé avec l'installeur pear, vous devrez confirmer que les librairies PHPUnit sont sur le `include_path` de PHP. Vous pouvez faire cela en vérifiant votre fichier php.ini et en vous assurant que les fichiers de PHPUnit sont dans un des répertoires de `include_path`.

Astuce : Toute sortie est swallowed lors de l'utilisation de PHPUnit 3.6+. Ajoutez le modificateur `--debug` si vous utiliser le CLI ou ajoutez `&debug=1` à l'url si vous utilisez le navigateur web pour afficher la sortie.

Tester la Configuration de la Base de Données

Souvenez-vous qu'il faut avoir un niveau de debug d'au moins 1 dans votre fichier `app/Config/core.php` avant de lancer des tests. Les tests ne sont pas accessibles via le navigateur quand le debug est égal à 0. Avant de lancer des tests, vous devrez vous assurer d'ajouter une configuration de base de données `$test`. Cette configuration est utilisée par CakePHP pour les tables fixture et les données :

```
public $test = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host'       => 'dbhost',
    'login'      => 'dblogin',
    'password'   => 'dbpassword',
    'database'   => 'test_database'
);
```

Note : C'est une bonne idée de faire une base de données de test différente de votre base de données actuelle. Cela évitera toute erreur embarrassante pouvant arriver plus tard.

Vérifier la Configuration Test

Après avoir installé PHPUnit et configuré le `$test` de la configuration de la base de données, vous pouvez vous assurer que vous êtes prêt à écrire et lancer vos propres tests en lancant un de ceux présents dans le coeur. Il y a deux exécuteurs integrés pour le test, nous commencerons en utilisant l'exécution par le navigateur. Les tests peuvent être accessibles par le navigateur à http ://localhost/votre_app/test.php. Vous devriez voir une liste des cas de test du coeur. Cliquez sur le test 'AllConfigure'. Vous devriez voir une barre verte avec quelques informations supplémentaires sur les tests lancés, et les nombres passés.

Félicitations, vous êtes maintenant prêt à commencer à écrire des tests !

Conventions des cas de Test

Comme beaucoup de choses dans CakePHP, les cas de test ont quelques conventions. En ce qui concerne les tests :

1. Les fichiers PHP contenant les tests devraient être dans votre répertoire `app/Test/Case/[Type]`.
2. Les noms de fichier de ces fichiers devraient finir avec `Test.php` à la place de .php.
3. Les classes contenant les tests devraient étendre `CakeTestCase`, `ControllerTestCase` ou `PHPUnit_Framework_TestCase`.
4. Comme les autres noms de classe, les noms de classe des cas de test doivent correspondre au nom de fichier. `RouterTest.php` doit contenir `class RouterTest extends CakeTestCase`.
5. Le nom de toute méthode contenant un test (par ex : contenant une assertion) devrait commencer par `test`, comme dans `testPublished()`. Vous pouvez aussi utiliser l'annotation `@test` pour marquer les méthodes en méthodes de test.

Quand vous avez crée un cas de test, vous pouvez l'exécuter en naviguant sur `http://localhost/votre_app/test.php` (selon votre configuration spécifique) Cliquez sur les cas de test de App, et cliquez ensuite sur le lien de votre fichier spécifique. Vous pouvez lancer les tests à partir des lignes de commande en utilisant le shell de test :

```
./Console/cake testsuite app Model/Post
```

Par exemple, lancerait les tests pour votre model Post.

Créer Votre Premier Cas de Test

Dans l'exemple suivant, nous allons créer un cas de test pour une méthode de helper très simple. Le helper que nous allons tester sera formaté en progress bar HTML. Notre helper ressemblerait à cela :

```
class ProgressHelper extends AppHelper {
    public function bar($value) {
        $width = round($value / 100, 2) * 100;
        return sprintf(
            '<div class="progress-container">
                <div class="progress-bar" style="width: %s%%"></div>
            </div>', $width);
    }
}
```

C'est un exemple très simple, mais ce sera utile pour montrer comment vous pouvez créer un cas de test simple. Après avoir créer et sauvegardé notre helper, nous allons créer le fichier de cas de tests dans `app/Test/Case/View/Helper/ProgressHelperTest.php`. Dans ce fichier, nous allons commencer avec ce qui suit :

```
App::uses('Controller', 'Controller');
App::uses('View', 'View');
App::uses('ProgressHelper', 'View/Helper');

class ProgressHelperTest extends CakeTestCase {
    public function setUp() {

    }

    public function testBar() {

    }
}
```

Nous compléterons ce squelette dans une minute. Nous avons ajouté deux méthodes pour commencer. Tout d'abord `setUp()`. Cette méthode est appelée avant chaque méthode de *test* dans une classe de cas de test. Les méthodes de configuration devraient initialiser les objets souhaités pour le test, et faire toute configuration souhaitée. Dans notre configuration nous ajouterons ce qui suit :

```
public function setUp() {
    parent::setUp();
    $Controller = new Controller();
```

```
    $View = new View($Controller);
    $this->Progress = new ProgressHelper($View);
}
```

Appeler la méthode parente est importante dans les cas de test, puisque CakeTestCase : :setUp() fait un nombre de choses comme fabriquer les valeurs dans `Configure` et, stocker les chemins dans `App`.

Ensuite, nous allons remplir les méthodes de test. Nous utiliserons quelques assertions pour nous assurer que notre code crée la sortie que nous attendions :

```
public function testBar() {
    $result = $this->Progress->bar(90);
    $this->assertContains('width: 90%', $result);
    $this->assertContains('progress-bar', $result);

    $result = $this->Progress->bar(33.3333333);
    $this->assertContains('width: 33%', $result);
}
```

Le test ci-dessus est simple mais montre le bénéfice potentiel de l'utilisation des cas de test. Nous utilisons `assertContains()` pour nous assurer que notre helper retourne une chaîne qui contient le contenu que nous attendons. Si le résultat ne contient pas le contenu attendu le test serait un échec, et saurait que notre code est incorrect.

En utilisant les cas de test, vous pouvez facilement décrire la relation entre un ensemble d'entrées connus et leur sortie attendue. Cela vous aide à être plus confiant sur le code que vous écrivez puisque vous pouvez facilement vérifier que le code que vous écrivez remplit les attentes et les assertions que vos tests font. De plus, puisque les tests sont du code, ils peuvent facilement être re-lancés dès que vous faîtes un changement. Cela évite la création de nouveaux bugs.

Lancer les Tests

Une fois que vous avez installé PHPUnit et que quelques cas de tests sont écrits, vous voudrez lancer les cas de test très fréquemment. C'est une bonne idée de lancer les tests avant de committer tout changement pour aider à s'assurer que vous n'avez rien cassé.

Lancer les tests à partir d'un navigateur

CakePHP fournit une interface web pour lancer les tests, donc vous pouvez exécuter vos tests par le navigateur si vous êtes plus habitué à cet environnement. Vous pouvez accéder au web runner en allant sur `http://localhost/votre_app/test.php`. La localisation exacte du test.php va changer en fonction de votre configuration. Mais le fichier est au même niveau que `index.php`.

Une fois que vous chargé les test runner, vous pouvez naviguer dans les suites test de App, Core et Plugin. Cliquer sur un cas de test individuel va lancer ce test et afficher les résultats.

Voir la couverture du code

Si vous avez XDebug (http ://xdebug.org) installé, vous pouvez voir les résultats de la couverture du code. La couverture du Code est utile pour vous dire quelles parties de votre code vos tests n'atteignent pas. La couverture est utile pour déterminer où vous devriez ajouter les tests dans le futur, et vous donne une mesure pour marquer la progression de vos tests.

```
337  * @param string $name
338  * @return void
339  */
340         public function __isset($name) {
341                 switch ($name) {
342                         case 'base':
343                         case 'here':
344                         case 'webroot':
345                         case 'data':
346                         case 'action':
347                         case 'params':
348                                 return true;
349                 }
350
351                 if (is_array($this->uses)) {
352                         foreach ($this->uses as $modelClass) {
353                                 list($plugin, $class) = pluginSplit($modelClass, true);
354                                 if ($name === $class) {
355                                         if (!$plugin) {
356                                                 $plugin = $this->plugin ? $this->plugin . '.' : null;
357                                         }
358                                         return $this->loadModel($modelClass);
359                                 }
360                         }
361                 }
362
363                 if ($name === $this->modelClass) {
364                         list($plugin, $class) = pluginSplit($name, true);
365                         if (!$plugin) {
366                                 $plugin = $this->plugin ? $this->plugin . '.' : null;
367                         }
368                         return $this->loadModel($plugin . $this->modelClass);
369                 }
370
371                 return false;
372         }
373
```

La couverture du code inline utilise les lignes vertes pour indiquer les lignes qui ont été exécutées. Si vous vous placez sur une ligne verte, une info-bulle indiquera quels tests couvre la ligne. Les lignes en rouge n'ont pas été lancées, et n'ont pas été testées par vos tests. Les lignes grises sont considerées comme du code non exécuté par xdebug.

Lancer les tests à partir d'une ligne de commande

CakePHP fournit un shell `test` pour lancer les tests. Vous pouvez lancer les tests de app, core et plugin facilement en utilisant le shell testsuite. Il accepte aussi tous les arguments que vous vous attendez à trouver sur l'outil de ligne de commnde du PHPUnit normal. A partir de votre répertoire app, vous pouvez faire ce qui suit pour lancer les tests :

```
# Lancer un test de model dans app
./Console/cake test app Model/Article

# Lancer un test de component dans un plugin
./Console/cake test DebugKit Controller/Component/ToolbarComponent
```

```
# Lancer le test de la classe de configuration dans CakePHP
./Console/cake test core Core/Configure
```

Note : Si vous lancez des tests qui intéragissent avec la session, c'est généralement une bonne idée d'utiliser l'option `--stderr`. Cela réglera les problèmes des échecs de test dûs aux avertissements des headers_sent.

Modifié dans la version 2.1 : Le shell `test` a été ajouté dans 2.1. Le shell `testsuite` de 2.0 est toujours disponible mais la nouvelle syntaxe est préférable. Vous pouvez aussi lancer le shell `test` dans le répertoire de projet racine. Cela vous montre une liste complète de tous les tests que vous avez actuellement. Vous pouvez ainsi choisir librement quel(s) test(s) lancer :

```
# Lancer test dans le réperoire de projet racine pour le dossier applicaton appelé app
lib/Cake/Console/cake test app

# Lancer test dans le repértoire de projets racine pour une application dans ./myapp
lib/Cake/Console/cake test -app myapp app
```

Filtrer les cas de test

Quand vous avez des cas de test plus larges, vous voulez souvent lancer un sous-ensemble de méthodes de test quand vous essayez de travailler sur un cas unique d'échec. Avec l'exécuteur cli vous pouvez utiliser une option pour filtrer les méthodes de test :

```
./Console/cake test core Console/ConsoleOutput --filter testWriteArray
```

Le paramètre filter est utilisé commme une expression régulière sensible à la casse pour filtrer les méthodes de test à lancer.

Générer une couverture de code

Vous pouvez générer un rapport de couverture de code à partir d'une ligne de commande en utilisant les outils de couverture de code intégrés dans PHPUnit. PHPUnit va générer un ensemble de fichiers en HTML statique contenant les résultats de la couverture. Vous pouvez générer une couverture pour un cas de test en faisant ce qui suit :

```
./Console/cake test app Model/Article --coverage-html webroot/coverage
```

Cela mettra la couverture des résultats dans le répertoire webroot de votre application. Vous pourrez voir les résultats en allant à `http://localhost/votre_app/coverage`.

Lancer les tests qui utilisent des sessions

Quand vous lancez des tests en ligne de commande qui utilisent des sessions, vous devrez inclure le flag `--stderr`. Ne pas le faire ne fera pas fonctionner les sessions. PHPUnit outputs test progress to stdout par défaut, cela entraine le fait que PHP suppose que les headers ont été envoyés ce qui empêche les sessions de démarrer. En changeant PHPUnit pour qu'il output on stderr, ce problème sera évité.

Les Callbacks du Cycle de vie des cas de Test

Les cas de Test ont un certain nombre de callbacks de cycle de vue que vous pouvez utiliser quand vous faîtes les tests :

- `setUp` est appelé avant chaque méthode de test. Doit être utilisé pour créer les objets qui vont être testés, et initialiser toute donnée pour le test. Toujours se rappeler d'appeler `parent::setUp()`.
- `tearDown` est appelé après chaque méthode de test. Devrait être utilisé pour nettoyer une fois que le test est terminé. Toujours se rappeler d'appeler `parent::tearDown()`.
- `setupBeforeClass` est appelé une fois avant que les méthodes de test aient commencées dans un cas. Cette méthode doit être *statique*.
- `tearDownAfterClass` est appelé une fois après que les méthodes de test ont commencé dans un cas. Cette méthode doit être *statique*.

Fixtures

Quand on teste du code qui dépend de models et d'une base de données, on peut utiliser les **fixtures** comme une façon de générer temporairement des tables de données chargées avec des données d'exemple qui peuvent être utilisées par le test. Le bénéfice de l'utilisation de fixtures est que votre test n'a aucune chance d'abimer les données de l'application qui tourne. De plus, vous pouvez commencer à tester votre code avant dee développer réellement en live le contenu pour une application.

CakePHP utilise la connection nommée `$test` dans votre fichier de configuration `app/Config/database.php` Si la connection n'est pas utilisable, une exception sera levée et vous ne serez pas capable d'utiliser les fixtures de la base de données.

CakePHP effectue ce qui suit pendant le chemin d'une fixture basée sur un cas de test :

1. Crée les tables pour chacun des fixtues necéssaires.
2. Remplit les tables avec les données, si les données sont fournis dans la fixture.
3. Lance les méthodes de test.
4. Vide les tables de fixture.
5. Retire les tables de fixture de la base de données.

Créer les fixtures

A la création d'une fixture, vous pouvez définir principalement deux choses : comment la table est créée (quels champs font parti de la table), et quels enregistrements seront remplis initialement dans la table. Créons notre première fixture, qui sera utilisée pour tester notre propre model Article. Crée un fichier nommé `ArticleFixture.php` dans votre répertoire `app/Test/Fixture` avec le contenu suivant :

```
class ArticleFixture extends CakeTestFixture {

      /* Optionel. Définir cette propriété pour charger les fixtures dans une source de do
      public $useDbConfig = 'test';
      public $fields = array(
          'id' => array('type' => 'integer', 'key' => 'primary'),
          'title' => array('type' => 'string', 'length' => 255, 'null' => false),
          'body' => 'text',
```

```
        'published' => array('type' => 'integer', 'default' => '0', 'null' => false),
        'created' => 'datetime',
        'updated' => 'datetime'
    );
    public $records = array(
        array('id' => 1, 'title' => 'First Article', 'body' => 'First Article Body', 'pub
        array('id' => 2, 'title' => 'Second Article', 'body' => 'Second Article Body', 'p
        array('id' => 3, 'title' => 'Third Article', 'body' => 'Third Article Body', 'pub
    );
}
```

La propriété `$useDbConfig` définit la source de données que la fixture va utiliser. Si votre application utilise plusieurs sources de données, vous devriez faire correspondre les fixtures avec les sources de données du model, mais préfixé avec `test_`. Par exemple, si votre model utilise la source de données `mydb`, votre fixture devra utiliser la source de données `test_mydb`. Si la connection `test_mydb` n'existe pas, vos models vont utiliser la source de données `test` par défaut. Les sources de données de fixture doivent être préfixées par `test` pour réduire la possibilité de trucher accidentellement toutes les données de votre application quand vous lancez des tests.

Nous utilisons `$fields` pour spécifier les champs qui feront parti de cette table, et comment ils sont définis. Le format utilisé pour définir ces champs est le même qu'utilisé avec `CakeSchema`. Les clés disponibles pour la définition de la table sont :

type

Type de données interne à CakePHP. Actuellement supportés :

- `string` : redirige vers `VARCHAR`.
- `text` : redirige vers `TEXT`.
- `integer` : redirige vers `INT`.
- `float` : redirige vers `FLOAT`.
- `datetime` : redirige vers `DATETIME`.
- `timestamp` : redirige vers `TIMESTAMP`.
- `time` : redirige vers `TIME`.
- `date` : redirige vers `DATE`.
- `binary` : redirige vers `BLOB`.

key Défini à `primary` pour que le champ soit en AUTO_INCREMENT, et une PRIMARY KEY pour la table.

length Défini à la longueur spécifique que le champ doit prendre.

null Défini soit à `true` (pour permettre les NULLs) soit à `false` (pour ne pas permettre les NULLs).

default Valeur par défaut que le champ prend.

Nos pouvons définir un ensemble d'enregistrements qui seront remplis après que la table de fixture est crée. Le format est directement fairly forward, `$records` est un tableau d'enregistrements. Chaque item dans `$records` devrait être une unique ligne. A l'intérieur de chaque ligne, il devrait y avoir un tableau associatif des colonnes et valeurs pour la ligne. Gardez juste à l'esprit que chaque enregistrement dans le tableau $records doit avoir une clé pour **chaque** champ spécifié dans le tableau `$fields`. Si un champ pour un enregistrement particulier a besoin d'avoir une valeur `null`, spécifiez juste la valeur de cette clé à `null`.

Les données dynamiques et les fixtures

Depuis que les enregistrements pour une fixture sont déclarées en propriété de classe, vous ne pouvez pas facilement utiliser les fonctions ou autres données dynamiques pour définir les fixtures. Pour résoudre ce problème, vous pouvez définir `$records` dans la fonction init() de votre fixture. Par exemple, si vous voulez tous les timestamps crées et mis à jours pour refléter la date d'aujourd'hui, vous pouvez faire ce qui suit :

```
class ArticleFixture extends CakeTestFixture {

    public $fields = array(
        'id' => array('type' => 'integer', 'key' => 'primary'),
        'title' => array('type' => 'string', 'length' => 255, 'null' => false),
        'body' => 'text',
        'published' => array('type' => 'integer', 'default' => '0', 'null' => false),
        'created' => 'datetime',
        'updated' => 'datetime'
    );

    public function init() {
        $this->records = array(
            array(
                'id' => 1,
                'title' => 'First Article',
                'body' => 'First Article Body',
                'published' => '1',
                'created' => date('Y-m-d H:i:s'),
                'updated' => date('Y-m-d H:i:s'),
            ),
        );
        parent::init();
    }
}
```

Quand vous surchargez `init()`, rappelez-vous juste de toujours appeler `parent::init()`.

Importer les informations de table et les enregistrements

Votre application peut avoir déjà des models travaillant avec des données réelles associées à eux, et vous pouvez décider de tester votre application avec ces données. Ce serait alors un effort dupliqué pour avoir à définir une définition de table et/ou des enregistrements sur vos fixtures. Heureusement, il y a une façon pour vous de définir cette définition de table et/ou d'enregistrements pour une fixture particulière venant d'un model existant ou d'une table existante.

Commençons par un exemple. Imaginons que vous ayez un model nommé Article disponible dans votre application (qui est lié avec une table nommée articles), on changerait le fixture donné dans la section précédente (`app/Test/Fixture/ArticleFixture.php`) en ce qui suit :

```
class ArticleFixture extends CakeTestFixture {
    public $import = 'Article';
}
```

Cette déclaration dit à la suite test d'importer la définition de votre table à partir de la table liée au model appelé Article. Vous pouvez utiliser tout model disponible dans votre application. La déclaration va seulement importer le schéma Article, et n'importe pas d'enregistrements. Pour importer les enregistrements, vous pouvez faire ce qui suit :

```
class ArticleFixture extends CakeTestFixture {
    public $import = array('model' => 'Article', 'records' => true);
}
```

Si d'un autre côté vous avez une table créée mais pas de model disponible pour elle, vous pouvez spécifier que votre import se fera en lisant l'information de la table à la place. Par exemple :

```
class ArticleFixture extends CakeTestFixture {
    public $import = array('table' => 'articles');
}
```

Va importer la définition de la table à partir de la table appelée 'articles' en utilisant la connection à la base de donnée CakePHP nommée 'default'. Si vous voulez utiliser une connection différente, utilisez :

```
class ArticleFixture extends CakeTestFixture {
    public $import = array('table' => 'articles', 'connection' => 'other');
}
```

Puisqu'on utilise votre connection à la base de données CakePHP, si il y a un préfixe de table déclaré, il sera automatiquement utilisé quand on récupère l'information de la table. Pour forcer la fixture et aussi importer ses enregistrements, changez l'importation en

```
class ArticleFixture extends CakeTestFixture {
    public $import = array('table' => 'articles', 'records' => true);
}
```

Vous pouvez naturellement importer la définition de votre table à partir d'un model/d'une table existante, mais vous avez vos enregistrements directement définis dans le fixture comme il a été montré dans la section précédente. Par exemple :

```
class ArticleFixture extends CakeTestFixture {
    public $import = 'Article';
    public $records = array(
        array('id' => 1, 'title' => 'First Article', 'body' => 'First Article Body', 'publi
        array('id' => 2, 'title' => 'Second Article', 'body' => 'Second Article Body', 'pub
        array('id' => 3, 'title' => 'Third Article', 'body' => 'Third Article Body', 'publi
    );
}
```

Charger les fixtures dans vos cas de test

Après avoir créé vos fixtures, vous voudrez les utiliser dans vos cas de test. Dans chaque cas de test vous devriez charger les fixtures dont vous aurez besoin. Vous devriez charger une fixture pour chaque model qui aura une requête lancée contre elle. Pour charger les fixtures, vous définissez la propriété `$fixtures` dans votre model :

```
class ArticleTest extends CakeTestCase {
    public $fixtures = array('app.article', 'app.comment');
}
```

Ce qui est au-dessus va charger les fixtures d'Article et de Comment à partir du répertoire de fixture de l'application. Vous pouvez aussi charger les fixtures à partir du coeur de CakePHP ou des plugins :

```
class ArticleTest extends CakeTestCase {
    public $fixtures = array('plugin.debug_kit.article', 'core.comment');
}
```

Utiliser le préfixe `core` va charger les fixtures à partir de CakePHP, et utiliser un nom de plugin en préfixe chargera le fixture à partir d'un plugin nommé.

Vous pouvez contrôler quand vos fixtures sont chargés en configurant `CakeTestCase::$autoFixtures` à `false` et plus tard les charger en utilisant `CakeTestCase::loadFixtures()` :

```
class ArticleTest extends CakeTestCase {
    public $fixtures = array('app.article', 'app.comment');
    public $autoFixtures = false;

    public function testMyFunction() {
        $this->loadFixtures('Article', 'Comment');
    }
}
```

Tester les Models

Disons que nous avons déjà notre model Article défini dans `app/Model/Article.php`, qui ressemble à ceci :

```
class Article extends AppModel {
    public function published($fields = null) {
        $params = array(
            'conditions' => array(
                $this->name . '.published' => 1
            ),
            'fields' => $fields
        );

        return $this->find('all', $params);
    }
}
```

Nous voulons maintenant configurer un test qui va utiliser la définition du model, mais à travers les fixtures, pour tester quelques fonctionnalités dans le model. Le test suite de CakePHP charge un petit ensemble minimum de fichiers (pour garder les test isolés), ainsi nous devons commencer par charger notre model - dans ce cas le model Article que nous avons déjà défini.

Créons maintenant un fichier nommé `ArticleTest.php` dans votre répertoire `app/Test/Case/Model`, avec les contenus suivants :

```
App::uses('Article', 'Model');

class ArticleTestCase extends CakeTestCase {
    public $fixtures = array('app.article');
}
```

Dans notre variable de cas de test `$fixtures`, nous définissons l'ensemble des fixtures que nous utiliserons. Vous devriez vous rappeler d'inclure tous les fixtures qui vont avoir des requêtes lancées contre elles.

Note : Vous pouvez écraser la base de données du model test en spécifiant la propriété `$useDbConfig`. Assurez-vous que la fixture utilise la même valeur afin que la table soit créee dans la bonne base de données.

Créer une méthode de test

Ajoutons maintenant une méthode pour tester la fonction published() dans le model Article. Modifier le fichier `app/Test/Case/Model/ArticleTest.php` afin qu'il ressemble maintenant à ceci :

```
App::uses('Article', 'Model');

class ArticleTest extends CakeTestCase {
    public $fixtures = array('app.article');

    public function setUp() {
        parent::setUp();
        $this->Article = ClassRegistry::init('Article');
    }

    public function testPublished() {
        $result = $this->Article->published(array('id', 'title'));
        $expected = array(
            array('Article' => array('id' => 1, 'title' => 'First Article')),
            array('Article' => array('id' => 2, 'title' => 'Second Article')),
            array('Article' => array('id' => 3, 'title' => 'Third Article'))
        );

        $this->assertEquals($expected, $result);
    }
}
```

Vous pouvez voir que nous avons ajouté une méthode appelée `testPublished()`. Nous commençons par créer une instance de notre model `Article`, et lançons ensuite notre méthode `published()`. Dans `$expected`, nous définissons ce que nous en attendons, ce qui devrait être le résultat approprié (que nous connaissons depuis que nous avons défini quels enregistrements sont remplis initialement dans la table articles.). Nous testons que les résultats correspondent à nos attentes en utilisant la méthode `assertEquals`. Regarder la section sur les *Lancer les Tests* pour plus d'informations sur la façon de lancer les cas de test.

Note : Quand vous configurez votre Model pour le test, assurez-vous d'utiliser `ClassRegistry::init('YourModelName');` puisqu'il sait comment utiliser la connection à la base de données de votre test.

Méthodes de Mocking des models

Il y aura des fois où vous voudrez mock les méhodes sur les models quand vous les testez. Vous devrez utiliser `getMockForModel` pour créer les mocks de test des models. Cela évite des problèmes avec les reflected properties that normal mocks have :

```
public function testSendingEmails() {
    $model = $this->getMockForModel('EmailVerification', array('send'));
    $model->expects($this->once())
        ->method('send')
        ->will($this->returnValue(true));

    $model->verifyEmail('test@example.com');
}
```

Introduit dans la version 2.3 : CakeTestCase : :getMockForModel() a été ajoutée dans 2.3.

Tester les Controllers

Alors que vous pouvez tester les classes de controller de la même manière que les Helpers, Models et Components, CakePHP offre une classe spécialisée `ControllerTestCase`. L'utilisation de cette classe en tant que classe de base pour les cas de test de votre controller vous permet d'utiliser `testAction()` pour des cas test plus simples. `ControllerTestCase` vous permet de facilement mock out les components et les models, ainsi que la difficulté potentielle pour tester les méthodes comme `redirect()`.

Disons que vous avez un controller typique Articles, et son model correspondant. Le code du controller ressemble à ceci :

```
class ArticlesController extends AppController {
    public $helpers = array('Form', 'Html');

    public function index($short = null) {
        if (!empty($this->request->data)) {
            $this->Article->save($this->request->data);
        }
        if (!empty($short)) {
            $result = $this->Article->findAll(null, array('id', 'title'));
        } else {
            $result = $this->Article->findAll();
        }

        if (isset($this->params['requested'])) {
            return $result;
        }
```

```
        $this->set('title', 'Articles');
        $this->set('articles', $result);
    }
}
```

Créez un fichier nommé `ArticlesControllerTest.php` dans votre répertoire `app/Test/Case/Controller` et mettez ce qui suit à l'intérieur :

```
class ArticlesControllerTest extends ControllerTestCase {
    public $fixtures = array('app.article');

    public function testIndex() {
        $result = $this->testAction('/articles/index');
        debug($result);
    }

    public function testIndexShort() {
        $result = $this->testAction('/articles/index/short');
        debug($result);
    }

    public function testIndexShortGetRenderedHtml() {
        $result = $this->testAction(
           '/articles/index/short',
            array('return' => 'contents')
        );
        debug($result);
    }

    public function testIndexShortGetViewVars() {
        $result = $this->testAction(
            '/articles/index/short',
            array('return' => 'vars')
        );
        debug($result);
    }

    public function testIndexPostData() {
        $data = array(
            'Article' => array(
                'user_id' => 1,
                'published' => 1,
                'slug' => 'new-article',
                'title' => 'New Article',
                'body' => 'New Body'
            )
        );
        $result = $this->testAction(
            '/articles/index',
            array('data' => $data, 'method' => 'post')
        );
        debug($result);
    }
```

```
}
```

Cet exemple montre quelques façons d'utiliser testAction pour tester vos controllers. Le premier paramètre de `testAction` devrait toujours être l'URL que vous voulez tester. CakePHP va créer une requête et dispatcher le controller et l'action.

Quand vous testez les actions qui contiennent `redirect()` et d'autres codes suivants le redirect, il est généralement bon de retourner quand il y a redirection. La raison pour cela est que `redirect()` est mocked dans les tests, et n'échappe pas comme à la normale. Et à la place de votre code existant, il va continuer de lancer le code suivant le redirect. Par exemple :

```
class ArticlesController extends AppController {
    public function add() {
        if ($this->request->is('post')) {
            if ($this->Article->save($this->request->data)) {
                $this->redirect(array('action' => 'index'));
            }
        }
        // plus de code
    }
}
```

Quand vous testez le code ci-dessus, vous allez toujours lancer `// plus de code` même si le redirect est atteint. A la place, vous devriez écrire le code comme ceci :

```
class ArticlesController extends AppController {
    public function add() {
        if ($this->request->is('post')) {
            if ($this->Article->save($this->request->data)) {
                return $this->redirect(array('action' => 'index'));
            }
        }
        // plus de code
    }
}
```

Dans ce cas `// plus de code` ne sera pas exécuté puisque la méthode retourne une fois que le redirect est atteint.

Simuler les requêtes GET

Comme vu dans l'exemple `testIndexPostData()` ci-dessus, vous pouvez utiliser `testAction()` pour tester les actions POST ainsi que les actions GET. En fournissant la clé `data`, la requête faite par le controller sera POST. Par défaut, toutes les requêtes seront des requêtes POST. Vous pouvez simuler une requête GET en configurant la méthode clé :

```
public function testAdding() {
    $data = array(
        'Post' => array(
            'title' => 'New post',
            'body' => 'Secret sauce'
```

```
        )
    );
    $this->testAction('/posts/add', array('data' => $data, 'method' => 'get'));
    // some assertions.
}
```

La clé data sera utilisée en paramètres de recherche de chaînes quand on va simuler une requête GET.

Choisir le type de retour

Vous pouvez choisir plusieurs façons pour inspecter le succès de l'action de votre controller. Chacun offre une manière différente de s'assurer que votre code fait ce que vous en attendez :
- `vars` Récupère l'ensemble des variables de vue.
- `view` Récupère la vue rendue, sans un layout.
- `contents` Récupère la vue rendue en incluant le layout.
- `result` Récupère la valeur de retour de l'action du controller. Utile pour tester les méthodes requestAction.

La valeur par défaut est `result`. Tant que votre type de retour n'est pas `result`, vous pouvez aussi accéder aux autres types de retour en propriétés dans les cas de test :

```
public function testIndex() {
    $this->testAction('/posts/index');
    $this->assertInternalType('array', $this->vars['posts']);
}
```

Utiliser mocks avec testAction

Il y aura des fois où vous voudrez remplacer les components ou les models avec soit des objets partiellement mocké, soit des objets complètement mockés. Vous pouvez faire ceci en utilisant `ControllerTestCase::generate()`. `generate()` fait le sale boulot afin de générer les mocks sur votre controller. Si vous décidez de générer un controller à utiliser dans les tests, vous pouvez générer les versions mockés de ses models et components avec ceci :

```
$Posts = $this->generate('Posts', array(
    'methods' => array(
        'isAuthorized'
    ),
    'models' => array(
        'Post' => array('save')
    ),
    'components' => array(
        'RequestHandler' => array('isPut'),
        'Email' => array('send'),
        'Session'
    )
));
```

Ce qui est au-dessus créerait un `PostsController` mocké, stubbing out la méthode `isAuthorized`. Le model Post attaché aura un `save()` stubbed, et les components attachés auront leurs méthodes respectives stubbed. Vous pouvez choisir de stub une classe entière en ne leur passant pas les méthodes, comme Session dans l'exemple ci-dessus.

Les controllers générés sont automatiquement utilisés en tant que controller de test à tester. Pour activer la génération automatique, définissez la variable `autoMock` dans le cas de test à true. Si `autoMock` est à false, votre controller original sera utilisé dans le test.

La réponse objet dans le controller généré est toujours remplacée par un mock qui n'envoie pas les headers. Après utilisation de `generate()` ou `testAction()`, vous pouvez accéder à l'objet controller à `$this->controller`.

Un exemple plus complexe

Dans sa plus simple forme, `testAction()` lancera `PostsController::index()` dans votre controller de test (ou en générera un automatiquement), en incluant tous les models mockés et les components. Les résultats du test sont stockés dans les propriétés `vars`, `contents`, `view`, et `return`. Une propriété headers est aussi disponible qui vous donne accès à `headers` qui aurait été envoyée, vous permettant de vérifier les redirects :

```
public function testAdd() {
    $Posts = $this->generate('Posts', array(
        'components' => array(
            'Session',
            'Email' => array('send')
        )
    ));
    $Posts->Session
        ->expects($this->once())
        ->method('setFlash');
    $Posts->Email
        ->expects($this->once())
        ->method('send')
        ->will($this->returnValue(true));

    $this->testAction('/posts/add', array(
        'data' => array(
            'Post' => array('name' => 'New Post')
        )
    ));

    $this->assertContains('/posts/index', $this->headers['Location']);
    $this->assertEquals('New Post', $this->vars['post']['Post']['name']);
    $this->assertRegExp('/<html/', $this->contents);
    $this->assertRegExp('/<form/', $this->view);
}
```

Cet exemple montre une utilisation légèrement plus complexe des méthodes `testAction()` et `generate()`. Tout d'abord, nous générons un controller de test et mock le `SessionComponent`. Maintenant que SessionComponent est mocké, nous avons la possibilité de lancer des méthodes de test dessus.

En supposant que `PostsController::add()` nous redirige à l'index, envoie un email et définit un message flash, le test va passer. Pour le bénéfice de l'exemple, nous vérifions aussi si le layout a été chargé en vérifiant les contenus entièrement rendus, et vérifions la vue pour un tag form. Comme vous pouvez le voir, votre liberté pour tester les controllers et facilement mocker ses classes est grandement étendue avec ces changements.

Quand vous faîtes des tests de controller en utilisant les mocks qui utilisent les méthodes statiques, vous devrez utiliser une méthode différente pour inscrire vos attentes de mock. Par exemple si vous voulez mock out `AuthComponent::user()` vous devrez faire ce qui suit :

```
public function testAdd() {
    $Posts = $this->generate('Posts', array(
        'components' => array(
            'Session',
            'Auth' => array('user')
        )
    ));
    $Posts->Auth->staticExpects($this->any())
        ->method('user')
        ->with('id')
        ->will($this->returnValue(2));
}
```

En utilisant `staticExpects` vous serez capable de mock et de manipuler les méthodes statiques sur les components et models.

Tester un Controller de Réponse JSON

JSON est un format sympa et courant à utiliser quand on construit un service web. Tester les endpoints de votre service web est très simple avec CakePHP. Commençons par un exemple de controller simple qui réponde dans JSON :

```
class MarkersController extends AppController {
    public $autoRender = false;
    public function index() {
        $data = $this->Marker->find('first');
        $this->response->body(json_encode($data));
    }
}
```

Maintenant nous créons le fichier `app/Test/Case/Controller/MarkersControllerTest.php` et nous assurons que notre service web retourne la réponse appropriée :

```
class MarkersControllerTest extends ControllerTestCase {
    public function testIndex() {
        $result = $this->testAction('/markers/index.json');
        $result = json_decode($result, true);
        $expected = array(
            'Marker' => array('id' => 1, 'lng' => 66, 'lat' => 45),
        );
        $this->assertEquals($expected, $result);
```

```
    }
}
```

Tester les Views

Généralement, la plupart des applications ne va pas directement tester leur code HTML. Faire ça donne souvent des résultats fragiles, il est difficile de maintenir les suites de test qui sont sujet à se casser. En écrivant des tests fonctionnels en utilisant `ControllerTestCase`, vous pouvez inspecter le contenu de la vue rendue en configurant l'option `return` à 'view'. Alors qu'il est possible de tester le contenu de la vue en utilisant ControllerTestCase, un test d'intégration/vue plus robuste et maintenable peut être effectué en utilisant des outils comme Selenium webdriver (http ://seleniumhq.org).

Tester les Components

Imaginons que nous avons un component appelé PagematronComponent dans notre application. Ce component nous aide à paginer la valeur limite à travers tous les controllers qui l'utilisent. Voici notre exemple de component localisé dans `app/Controller/Component/PagematronComponent.php` :

```
class PagematronComponent extends Component {
    public $Controller = null;

    public function startup(Controller $controller) {
        parent::startup($controller);
        $this->Controller = $controller;
        // Assurez-vous que le controller utilise la pagination
        if (!isset($this->Controller->paginate)) {
            $this->Controller->paginate = array();
        }
    }

    public function adjust($length = 'short') {
        switch ($length) {
            case 'long':
                $this->Controller->paginate['limit'] = 100;
            break;
            case 'medium':
                $this->Controller->paginate['limit'] = 50;
            break;
            default:
                $this->Controller->paginate['limit'] = 20;
            break;
        }
    }
}
```

Maintenant nous pouvons écrire des tests pour nous assurer que notre paramètre de pagination `limit` est défini correctement par la méthode `adjust` dans notre component. Nous créons le fichier `app/Test/Case/Controller/Component/PagematronComponentTest.php` :

```
App::uses('Controller', 'Controller');
App::uses('CakeRequest', 'Network');
App::uses('CakeResponse', 'Network');
App::uses('ComponentCollection', 'Controller');
App::uses('PagematronComponent', 'Controller/Component');

// Un faux controller pour tester against
class TestPagematronController extends Controller {
    public $paginate = null;
}

class PagematronComponentTest extends CakeTestCase {
    public $PagematronComponent = null;
    public $Controller = null;

    public function setUp() {
        parent::setUp();
        // Configurer notre component et faire semblant de tester le controller
        $Collection = new ComponentCollection();
        $this->PagematronComponent = new PagematronComponent($Collection);
        $CakeRequest = new CakeRequest();
        $CakeResponse = new CakeResponse();
        $this->Controller = new TestPagematronController($CakeRequest, $CakeResponse);
        $this->PagematronComponent->startup($this->Controller);
    }

    public function testAdjust() {
        // Tester notre méthode adjust avec les configuraitons de différents paramètres
        $this->PagematronComponent->adjust();
        $this->assertEquals(20, $this->Controller->paginate['limit']);

        $this->PagematronComponent->adjust('medium');
        $this->assertEquals(50, $this->Controller->paginate['limit']);

        $this->PagematronComponent->adjust('long');
        $this->assertEquals(100, $this->Controller->paginate['limit']);
    }

    public function tearDown() {
        parent::tearDown();
        // Nettoie après l'avoir fait
        unset($this->PagematronComponent);
        unset($this->Controller);
    }
}
```

Tester les Helpers

Puisqu'un bon nombre de logique se situe dans les classes Helper, il est important de s'assurer que ces classes sont couvertes par des cas de test.

Tout d'abord, nous créons un helper d'exemple à tester. `CurrencyRendererHelper` va nous aider à

afficher les monnaies dans nos vues et pour siplifier, il ne va avoir qu'une méthode `usd()`.

```
// app/View/Helper/CurrencyRendererHelper.php
class CurrencyRendererHelper extends AppHelper {
    public function usd($amount) {
        return 'USD ' . number_format($amount, 2, '.', ',');
    }
}
```

Ici nous définissons la décimale à 2 après la virgule, le séparateur de décimal, le séparateur des centaines avec une virgule, et le nombre formaté avec la chaîne 'USD' en préfixe.

Maintenant nous créons nos tests :

```
// app/Test/Case/View/Helper/CurrencyRendererHelperTest.php

App::uses('Controller', 'Controller');
App::uses('View', 'View');
App::uses('CurrencyRendererHelper', 'View/Helper');

class CurrencyRendererHelperTest extends CakeTestCase {
    public $CurrencyRenderer = null;

    // Ici nous instancions notre helper
    public function setUp() {
        parent::setUp();
        $Controller = new Controller();
        $View = new View($Controller);
        $this->CurrencyRenderer = new CurrencyRendererHelper($View);
    }

    // Test de la fonction usd()
    public function testUsd() {
        $this->assertEquals('USD 5.30', $this->CurrencyRenderer->usd(5.30));

        // Nous devrions toujours avoir 2 décimales
        $this->assertEquals('USD 1.00', $this->CurrencyRenderer->usd(1));
        $this->assertEquals('USD 2.05', $this->CurrencyRenderer->usd(2.05));

        // Test du séparateur des milliers
        $this->assertEquals('USD 12,000.70', $this->CurrencyRenderer->usd(12000.70));
    }
}
```

Ici nous appelons `usd()` avec des paramètres différents et disons à test suite de vérifier si les valeurs retournées sont égales à ce que nous en attendons.

Sauvegardons cela et exécutons le test. Vous devriez voir une barre verte et un message indiquant 1 passé et 4 assertions.

Créer les Test Suites

Si vous voulez que plusieurs de vos tests soient lancés en même temps vous pouvez créer un test suite. Un testsuite est composé de plusieurs cas de test. `CakeTestSuite` offre quelques méthodes pour facilement créer des test suites basé sur le système de fichier. Si nous voulions créer un test suite pour tous nos models tests, nous pourrions créer `app/Test/Case/AllModelTest.php`. Mettez ce qui suit dedans :

```
class AllModelTest extends CakeTestSuite {
    public static function suite() {
        $suite = new CakeTestSuite('All model tests');
        $suite->addTestDirectory(TESTS . 'Case' . DS . 'Model');
        return $suite;
    }
}
```

Le code ci-dessus va grouper tous les cas de test trouvés dans le dossier `/app/Test/Case/Model/`. Pour ajouter un fichier individuel, utilisez `$suite->addTestFile($filename);`. Vous pouvez ajouter de façon récursive un répertoire pour tous les tests en utilisant :

```
$suite->addTestDirectoryRecursive(TESTS . 'Case');
```

Ajouterait de façon récursive tous les cas de test dans le répertoire `app/Test/Case/`.

Créer des Tests pour les Plugins

Les Tests pour les plugins sont crées dans leur propre répertoire à l'intérieur du dossier des plugins. :

```
/app
    /Plugin
        /Blog
            /Test
                /Case
                /Fixture
```

Ils travaillent comme des tests normaux mais vous devrez vous souvenir d'utiliser les conventions de nommage pour les plugins quand vous importez des classes. Ceci est un exemple d'un testcase pour le model `BlogPost` à partir du chapitre des plugins de ce manuel. Une différence par rapport aux autres test est dans la première ligne où 'Blog.BlogPost' est importé. Vous devrez aussi préfixer les fixtures de votre plugin avec `plugin.blog.blog_post` :

```
App::uses('BlogPost', 'Blog.Model');

class BlogPostTest extends CakeTestCase {

    // Les fixtures de plugin localisé dans /app/Plugin/Blog/Test/Fixture/
    public $fixtures = array('plugin.blog.blog_post');
    public $BlogPost;

    public function testSomething() {
        // ClassRegistry dit au model d'utiliser la connection à la base de données test
        $this->BlogPost = ClassRegistry::init('Blog.BlogPost');
```

```
        // faire des tests utiles ici
        $this->assertTrue(is_object($this->BlogPost));
    }
}
```

Si vous voulez utiliser les fixures de plugin dans les app tests, vous pouvez y faire référence en utilisant la syntaxe `plugin.pluginName.fixtureName` dans le tableau `$fixtures`.

Intégration avec Jenkins

Jenkins (http ://jenkins-ci.org) est un serveur d'intégration continu, qui peut vous aider à automatiser l'exécution de vos cas de test. Cela aide à s'assurer que tous les tests passent et que votre application est déjà prête.

Intégrer une application CakePHP avec Jenkins est fairly straightforward. Ce qui suit suppose que vous avez déjà installé Jenkins sur un système *nix, et que vous êtes capable de l'administrer. Vous savez aussi comment créer des jobs, et lancer des builds. Si vous n'êtes pas sur de tout cela, réferez vous à la documentation de Jenkins (http ://jenkins-ci.org/).

Créer un job

Commençons par créer un job pour votre application, et connectons votre répertoire afin que jenkins puisse accéder à votre code.

Ajouter une config de base de données de test

Utiliser une base de données séparée juste pour Jenkins est généralement une bonne idée, puisque cela évite au sang de couler et évite un certain nombre de problèmes basiques. Une fois que vous avez crée une nouvelle base de données dans un serveur de base de données auquel jenkins peut accéder (habituellement localhost). Ajoutez une *étape de script shell* au build qui contient ce qui suit :

```
cat > app/Config/database.php <<'DATABASE_PHP'
<?php
class DATABASE_CONFIG {
    public $test = array(
        'datasource' => 'Database/Mysql',
        'host'       => 'localhost',
        'database'   => 'jenkins_test',
        'login'      => 'jenkins',
        'password'   => 'cakephp_jenkins',
        'encoding'   => 'utf8'
    );
}
DATABASE_PHP
```

Cela s'assure que vous aurez toujours la bonne configuration de la base de données dont Jenkins a besoin. Faites la même chose pour tout autre fichier de configuration dont vous auriez besoin. Il est souvent une bonne idée de supprimer et re-créer la base de données avant chaque build aussi. Cela vous évite des echecs

de chaînes, où un buid cassé entraîne l'echec des autres. Ajoutez une autre *étape de script shell* au build qui contient ce qui suit :

```
mysql -u jenkins -pcakephp_jenkins -e 'DROP DATABASE IF EXISTS jenkins_test; CREATE DATABAS
```

Ajouter vos tests

Ajoutez une autre *étape de script shell* à votre build. Dans cette étape, lancez les tests pour votre application. Créer un fichier de log junit, ou clover coverage est souvent un bonus sympa, puisqu'il vous donne une vue graphique sympa des résultats de votre test :

```
app/Console/cake testsuite app AllTests \
--stderr \
--log-junit junit.xml \
--coverage-clover clover.xml
```

Si vous utilisez le clover coverage, ou les résultats junit, assurez-vous de les configurer aussi dans Jenkins. Ne pas configurer ces étapes signifiera que vous ne verrez pas les résultats.

Lancer un build

Vous devriez être capable de lancer un build maintenant. Vérifiez la sortie de la console et faites tous les changements necéssaires pour obtenir un build précédent.

REST

Quelques programmeurs néophytes d'application réalisent le besoin d'ouvrir leurs fonctionnalités du coeur à un public plus important. Fournir facilement, un accès sans entrave à votre API du coeur peut aider à ce que votre plateforme soit acceptée, et permettre les mashups et une intégration facile avec les autres systèmes.

Alors que d'autres solutions existent, REST est un bon moyen de fournir facilement un accès à la logique que vous avez créee dans votre application. C'est simple, habituellement basé sur XML (nous parlons de XML simple, rien de semblable à une enveloppe SOAP), et dépend des headers HTTP pour la direction. Exposer une API via REST dans CakePHP est simple.

Mise en place Simple

Le moyen le plus rapide pour démarrer avec REST est d'ajouter quelques lignes à votre fichier routes.php, situé dans app/Config. L'objet Router comporte une méthode appelée `mapResources()` utilisée pour mettre en place un certain nombre de routes par défaut accédant par REST à vos controllers. Assurez-vous que `mapResources()` vienne avant `require CAKE . 'Config' . DS . 'routes.php';`. Si nous souhaitions permettre l'accès par REST à une base de données de recettes, nous ferions comme cela :

```
//Dans app/Config/routes.php...

Router::mapResources('recipes');
Router::parseExtensions();
```

La première ligne met en place un certain nombre de routes par défaut pour un accès facile par REST où la méthode spécifie le format de résultat souhaité (ex : xml, json, rss). Ces routes correspondent aux méthodes de requêtes HTTP.

HTTP format	URL.format	Action de contrôleur appelée
GET	/recipes.format	RecipesController : :index()
GET	/recipes/123.format	RecipesController : :view(123)
POST	/recipes.format	RecipesController : :add()
PUT	/recipes/123.format	RecipesController : :edit(123)
DELETE	/recipes/123.format	RecipesController : :delete(123)
POST	/recipes/123.format	RecipesController : :edit(123)

La classe Router de CakePHP utilise un certain nombre d'indicateurs différents pour détecter la méthode HTTP utilisée. Les voici par ordre de préférence :

1. La variable POST *_method*
2. X_HTTP_METHOD_OVERRIDE
3. Le header REQUEST_METHOD

La *_method* variable POST est utile lors de l'utilisation d'un navigateur en tant que client REST (ou n'importe quoi d'autre capable de faire facilement du POST). Il suffit d'initialiser la valeur de _method au nom de la méthode de requête HTTP que vous souhaitez émuler.

Une fois que le router est paramétré pour faire correspondre les requêtes REST à certaines actions de controller, nous pouvons nous mettre à créer la logique dans nos actions de controller. Un controller basique pourrait ressembler à ceci :

```
// Controller/RecipesController.php
class RecipesController extends AppController {

    public $components = array('RequestHandler');

    public function index() {
        $recipes = $this->Recipe->find('all');
        $this->set(array(
            'recipes' => $recipes,
            '_serialize' => array('recipes')
        ));
    }

    public function view($id) {
        $recipe = $this->Recipe->findById($id);
        $this->set(array(
            'recipe' => $recipe,
            '_serialize' => array('recipe')
        ));
    }
```

```
    public function edit($id) {
        $this->Recipe->id = $id;
        if ($this->Recipe->save($this->request->data)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
        $this->set(array(
            'message' => $message,
            '_serialize' => array('message')
        ));
    }

    public function delete($id) {
        if ($this->Recipe->delete($id)) {
            $message = 'Deleted';
        } else {
            $message = 'Error';
        }
        $this->set(array(
            'message' => $message,
            '_serialize' => array('message')
        ));
    }
}
```

Depuis que nous avons ajouté un appel à `Router::parseExtensions()`, Le router CakePHP est déjà prêt à servir différentes vues sur la base de différents types de requêtes. Puisque nous avons affaire à des requêtes REST, le type de vue est le XML. Vous pouvez aussi facilement faire des vues JSON en utilisant le *Vues JSON et XML* intégré dans CakePHP. En utilisant le `XmlView` intégré, nous pouvons définir une variable de vue `_serialize`. Cette variable de vue spéciale est utilisée pour définir quelles variables de vue `XmlView` devrait sérialiser dans XML.

Si vous souhaitez modifier les données avant d'être converties en XML, nous ne devrions pas `_serialize` une variable de vue, et à la place utiliser les fichiers de vue. Nous plaçons les vues REST pour nos RecipesController à l'intérieur de `app/View/recipes/xml`. Nous pouvons aussi utiliser `Xml` pour une sortie XML facile et rapide dans ces vues. Voici ce à quoi notre index pourrait ressembler :

```
// app/View/Recipes/xml/index.ctp
// Faire du formatage et des manipulations sur
// le tableau $recipes.
$xml = Xml::fromArray(array('response' => $recipes));
echo $xml->asXML();
```

Quand on sert un type de contenu spécifique en utilisant parseExtensions(), CakePHP recherche automatiquement un helper de vue qui correspond au type. Puisque nous utilisons XML en type de contenu, il n'y a pas de helper intégré, cependant si vous en créez un, il sera automatiquement charger pour notre utilisation dans ces vues.

Le XML rendu va au final ressembler à ceci :

```
<recipes>
    <recipe id="234" created="2008-06-13" modified="2008-06-14">
        <author id="23423" first_name="Billy" last_name="Bob"></author>
        <comment id="245" body="Yummy yummmy"></comment>
    </recipe>
    <recipe id="3247" created="2008-06-15" modified="2008-06-15">
        <author id="625" first_name="Nate" last_name="Johnson"></author>
        <comment id="654" body="This is a comment for this tasty dish."></comment>
    </recipe>
</recipes>
```

Créer la logique pour l'action edit est un peu vicieux, mais pas de beaucoup. Puisque nous fournissons un API qui sort du XML, c'est un chois naturel pour recevoir le XML en entrée. Ne vous inquiétez pas, les classes `RequestHandler` et `Router` facilitent beaucoup les choses. Si une requête POST ou PUT a un content-type XML, alors l'entrée est lancée à travers la classe `Xml` de Cake, et la représentation en tableau des données est assignée à *$this->request->data*. A cause de cette fonctionnalité, gérer les données en XML et POST en parallèle est transparente : aucun changement n'est requis dans le code du controller ou du model. Tout ce dont vous avez besoin devrait finir dans `$this->request->data`.

Accepter une entrée dans d'autres formats

Typiquement, les applications REST ne sortent pas seulement le contenu dans les formats de données alternatifs, elles acceptent aussi les données dans des formats différents. Dans CakePHP, `RequestHandlerComponent` facilite cela. Par défaut, il va décoder toute entrée de données entrante JSON/XML pour les requêtes POST/PUT et fournir la version de tableau de cette donnée dans *$this->request->data*. Vous pouvez aussi connecter dans les deserializers supplémentaires pour des formats alternatifs si vous en avez besoin, utilisez `RequestHandler::addInputType()`.

Modifier les routes REST par défaut

Introduit dans la version 2.1. Si les routes REST par défaut ne fonctionnent pas pour votre application application, vous pouvez les modifier en utilisant `Router::resourceMap()`. Cette méthode vous permet de définir les routes par défaut qui récupèrent l'ensemble avec `Router::mapResources()`. Quand vous utilisez cette méthode vous devez définir *toutes* les valeurs par défaut que vous voulez utiliser :

```
Router::resourceMap(array(
    array('action' => 'index', 'method' => 'GET', 'id' => false),
    array('action' => 'view', 'method' => 'GET', 'id' => true),
    array('action' => 'add', 'method' => 'POST', 'id' => false),
    array('action' => 'edit', 'method' => 'PUT', 'id' => true),
    array('action' => 'delete', 'method' => 'DELETE', 'id' => true),
    array('action' => 'update', 'method' => 'POST', 'id' => true)
));
```

En écrivant par dessus la ressource map par défaut, les appels futurs à `mapResources()` vont utiliser les nouvelles valeurs.

Routing REST Personnalisé

Si les routes créees par défaut par `Router::mapResources()` ne fonctionnent pas pour vous, utilisez la méthode `Router::connect()` pour définir un ensemble personnalisé de routes REST. La méthode `connect()` vous permet de définir un certain nombre d'options différentes pour une URL donnée. Regardez la section sur *Utiliser des conditions supplémentaires de correspondance des routes* pour plus d'informations.

Filtres du Dispatcher

Introduit dans la version 2.2. Il y a plusieurs raisons de vouloir un bout de code à lancer avant que tout code de controller soit lancé ou juste avant que la réponse soit envoyée au client, comme la mise en cache de la réponse, le header tuning, l'authentication spéciale ou juste pour fournir l'accès à une réponse de l'API critique plus rapidement qu'avec un cycle complet de dispatchement de requêtes.

CakePHP fournit une interface propre et extensible pour de tels cas pour attacher les filtres au cycle de dispatchement, de la même façon qu'une couce middleware pour fournir des services empilables ou des routines pour chaque requête. Nous les appelons *Dispatcher Filters*.

Configurer les Filtres

Les filtres sont généralement configurés dans le fichier `bootstrap.php`, mais vous pouvez facilement les charger à partir d'un autre fichier de configuration avant que la requête soit dispatchée. Ajouter et retirer les filtres est fait par la classe *Configure*, en utilisant la clé spéciale `Dispatcher.filters`. Par défaut CakePHP est fourni avec des classes de filtre déjà activées pour toutes les requêtes, allons voir comment elles ont été ajoutées :

```
Configure::write('Dispatcher.filters', array(
    'AssetDispatcher',
    'CacheDispatcher'
));
```

Chacune de ces valeurs de tableaux sont des noms de classe qui seront instanciés et ajoutés en écouteurs des évènements générés au niveau du dispatcher. Le premier, `AssetDispatcher` est là pour vérifier si la requête se réfère au theme ou au fichier d'asset du plugin, comme le css, le javascript ou une image stockée dans soit un dossier du webroot du plugin, soit dans celui correspondant au Theme. Il servira le fichier selon s'il est trouvé ou non, stoppant le reste du cycle de dispatchement. Le filtre `CacheDispatcher`, quand la variable de config `Cache.check` est activée, va vérifier si la réponse a déjà été mise en cache dans le système de fichier pour une requête similaire et servir le code mis en cache immédiatement.

Comme vous pouvez le voir, les deux filtres fournis ont la responsabilité d'arrêter tout code plus loin et d'envoyer la réponse juste après au client. Mais les filtres ne sont pas limités au role, puisque nous allons montrons rapidement dans cette section.

Vous pouvez ajouter vos propres noms de classes à la liste des filtres, et ils seront exécutés dans l'ordre dans lequel ils ont été défini. Il y aussi une façon alternative pour attacher les filtres qui n'impliquent pas les classes spéciales `DispatcherFilter` :

```
Configure::write('Dispatcher.filters', array(
    'my-filter' => array('callable' => array($classInstance, 'methodName'), 'on' => 'after'
));
```

Comme montré ci-dessus, vous pouvez passer tout type de callback (http ://php.net/callback) PHP valide, puisque vous vous en souvenez peut-être, un *callback* est tout ce que PHP peut exécuter avec `call_user_func`. Nous faisons une petite exception, si une chaîne est fournie, elle sera traitée comme un nom de classe, et pas comme un nom de fonction possible. Ceci bien sur donne la capacité aux utilisateurs de PHP5.3 d'attacher des fonctions anonymes en tant que filtres :

```
Configure::write('Dispatcher.filters', array(
   'my-filter' => array('callable' => function($event) {...}, 'on' => 'before'),
   //plus de filtres ici
));
```

La clé `on` prend seulement `before` et `after` comme valeurs valides, et cela signifie bien évidemment si le filtre doit être lancé avant ou après que tout code du controller soit executé. En plus de définir les filtres avec la clé `callable`, vous pouvez aussi définir une priorité pour vos filtres, si aucun n'est spécifié alors par défaut `10` et séléctionné pour vous.

Puisque tous les filtres auront une priorité par défaut à `10`, vous aurez envie de lancer un filtre avant tout autre dans la liste, sélectionner des nombres d'une priorité plus faible comme souhaité :

```
Configure::write('Dispatcher.filters', array(
   'my-filter' => array(
        'callable' => function($event) {...},
        'on' => 'before',
        'priority' => 5
    ),
    'other-filter' => array(
        'callable' => array($class, 'method'),
        'on' => 'after',
        'priority' => 1
    ),
   //plus de filtres ici
));
```

Evidemment, quand vous définissez les priorités, l'ordre dans lequel les filtres sont déclarés n'a pas d'importance mais pour ceux qui ont la même priorité. Quand vous définissez les filtres en tant que noms de classe, il n'y a pas d'options pour définir la priorité in-line, nous y viendront bientôt. Au final, la notation de plugin de CakePHP peut être utilisée pour définir les filtres localisés dans les plugins :

```
Configure::write('Dispatcher.filters', array(
    'MyPlugin.MyFilter',
));
```

N'hésitez pas à retirer les filtres attachés par défaut si vous choisissiez d'utiliser une façon plus avancée/rapide pour servir les thèmes les assets de plugin ou si vous ne souhaitez pas utiliser la mise en cache intégrée de la page entière, ou pour juste implémenter le votre.

Classes Filter

Les filtres de Dispatcher, quand définis en tant que noms de classe dans configuration, doivent étendre la classe `DispatcherFilter` fournie dans le répertoire *Routing* de CakePHP. Créeons un simple filtre pour répondre à une url spécifique avec un texte 'Hello World' :

```
App::uses('DispatcherFilter', 'Routing');
class HelloWorldFilter extends DispatcherFilter {

    public $priority = 9;

    public function beforeDispatch($event) {
        $request = $event->data['request'];
        $response = $event->data['response'];

        if ($request->url === 'hello-world') {
            $response->body('Hello World');
            $event->stopPropagation();
            return $response;
        }
    }
}
```

Cette classe devrait être sauvegardée dans un fichier dans `app/Routing/Filter/HelloWorldFilter.php` et configurée dans le fichier bootstrap comme on l'a expliqué dans la section précédente. Il y a plein de choses à expliquer ici, commençons avec la valeur `$priority`.

Comme mentionné avant, quand vous utilisez les classes de filtre, vous pouvez seulement définir l'ordre dans lequel elles sont lancées en utilisant la propriété `$priority` dans la classe, la valeur par défaut est 10 si la propriété est déclarée, cela signifie qu'il sera executé _après_ que la classe de Router a parsé la requête. Nous ne voulons pas que cela arrive dans notre exemple précédent, parce que probablement, vous n'avez pas de controller configuré pour répondre à cette url, donc nous avons choisi 9 comme notre priorité.

`DispatcherFilter` propose deux méthodes qui peuvent être écrasées dans des sous-classes, elles sont `beforeDispatch` et `afterDispatch`, et sont executées respectivement avant ou après que tout controller soit executé. Les deux méthodes reçoivent un objet `CakeEvent` contenant les objets `request` et `response` (instances `CakeRequest` et `CakeResponse`) avec un tableau `additionalParams` à l'intérieur de la propriété `data`. Ce qui suit contient des informations utilisées pour le dispatching interne quand on appelle `requestAction`.

Dans notre exemple, nous avons retourné selon les cas l'objet `$response` comme résultat, cela dira au Dispatcher pour n'instancier aucun controller et retourner un objet comme cela en réponse immédiatement au client. Nous avons aussi ajouté `$event->stopPropagation()` pour empêcher d'autres filtres d'être exécuté après celui-ci.

Créeons maintenant un autre filtre pour modifier les headers de réponse dans toute page publique, dans notre cas, ce serait tout ce qui est servi à partir de `PagesController` :

```
App::uses('DispatcherFilter', 'Routing');
class HttpCacheFilter extends DispatcherFilter {

    public function afterDispatch($event) {
```

```
        $request = $event->data['request'];
        $response = $event->data['response'];

        if ($request->params['controller'] !== 'pages') {
            return;
        }
        if ($response->statusCode() === 200) {
            $response->sharable(true);
            $response->expires(strtotime('+1 day'));
        }
    }
}
```

Ce filtre enverra une expiration du header à 1 jour dans le futur pour toutes les réponses produites par le controller pages. Vous pourriez bien sûr faire la même chose dans le controller, ceci est juste un exemple de ce qui peut être fait avec les filtres. Par exemple, au lieu de modifier la réponse, vous pourriez la mettre en cache en utilisant la classe `Cache` et servir la réponse à partir du callback `beforeDispatch`.

Filtres Inline

Notre dernier exemple va utiliser une fonction anonyme (seulement disponible sur PHP 5.3+) pour servir une liste de posts dans un format json, nous vous encourageons à faire ainsi l'utilisation des controllers et la classe `JsonView`, mais imaginons que vous ayez besoin de gagner une tout petite milliseconde pour cette mission-critical API endpoint :

```
$postsList = function($event) {
    if ($event->data['request']->url !== 'posts/recent.json') {
        return;
    }
    App::uses('ClassRegistry', 'Utility');
    $postModel = ClassRegistry::init('Post');
    $event->data['response']->body(json_encode($postModel->find('recent')));
    $event->stopPropagation();
    return $event->data['response'];
};

Configure::write('Dispatcher.filters', array(
    'AssetDispatcher',
    'CacheDispatcher',
    'recent-posts' => array(
        'callable' => $postsList,
        'priority' => 9,
        'on'=> 'before'
    )
));
```

Dans l'exemple précédent, nous avons selectionné une priorité à `9` pour notre filtre, donc pour sauter toute autre logique, soit placé dans des filtres personnalisés, soit dans des filtres du coeur comme le système de routing interne de CakePHP. Bien que cela ne soit pas necéssaire, cela montre comment faire pour que votre code important se lance en premier au cas où vous auriez besoin de trim au plus gros possible à partir de certaines requêtes.

Pour des raisons évidentes, ceci a le potentiel de rendre la maintenance de votre app très difficile. Les filtres sont un outil extrèmement puissant quand on les utilise sagement, ajoutez les gestionnaires de réponse pour chaque url dans votre app n'est pas une bonne utilisation pour cela. Mais si vous avez une raison valide de le faire, alors vous avez une solution propre à portée de main. Gardez à l'esprit que tout ne doit pas être un filtre, les *Controllers* et les *Components* sont habituellement un choix plus précis pour ajouter tout code de gestion de requête à votre app.

Les packages Vendor

Le fichier d'information sur Vendor ira ici.

Vendor assets

Le support pour les assets vendor a été retiré pour 1.3. Il est recommandé que vous preniez tout vendor asset que vous avez et que vous le repackagiez en plugins. Regardez *Plugin assets* pour plus d'informations.

CHAPITRE 11

Déploiement

Une fois que votre application est terminée, ou même avant que vous souhaitiez la déployer. Il y a quelques choses que vous devriez faire quand vous déployez une application CakePHP.

Vérifier votre sécurité

Si vous sortez votre application dans la nature, il est bon de vous assurer qu'elle n'a pas de fuites. Allez voir *Security (Sécurité)* pour vous sécuriser contre les attaques CSRF, form field tampering, etc... Utiliser *Validation des Donnnées*, et/ou *Data Sanitization (Nettoyage des Données)* est aussi une bonne idée, pour protéger votre base de données et aussi contre les attaques XSS. Vérifiez que seul votre répertoire `webroot` est visible publiquement, et que vos secrets (comme le salt de votre app, et les clés de sécurité) sont privés et uniques aussi !

Définir le document root

Configurer le document root correctement dans votre application est aussi une étape importante pour garder votre code sécurisé et votre application plus sûre. Les applications CakePHP devraient avoir le document root configuré au répertoire `app/webroot` de l'application. Cela rend les fichiers de l'application et de configuration inaccessibles via une URL. Configurer le document root est différent selon les webserveurs. Regardez la documentation *URL Rewriting* pour une information sur la spécificité de chaque webserveur.

Dans tous les cas, vous devez définir le document de l'hôte/domaine virtuel pour qu'il soit `app/webroot/`. Cela retire la possibilité que des fichiers soient executés en-dehors du répertoire webroot.

Mise à jour de core.php

Mettre à jour core.php, spécialement la valeur de `debug` est extrêmement important. Mettre debug = 0 désactive un certain nombre de fonctionnalités de développement qui ne devraient jamais être exposées sur internet. Désactiver le debug change les types de choses suivantes :

- Les messages de Debug, créés avec `pr()` et `debug()` sont désactivés.
- Les caches du Coeur de CakePHP sont flushés tous les 999 jours, au lieu de tous les 10 seconds en développement.
- Les vues d'Erreur sont moins informatives, et renvoient des messages génériques d'erreur à la place.
- Les Erreurs ne sont pas affichées.
- Les traces de pile d'Exception sont désactivées.

En plus des éléments ci-desssus, beaucoup de plugins et d'extensions d'application utilisent `debug` pour modifier leur comportement.

Améliorer les performances de votre application

Comme la gestion des éléments statiques, comme les images, le Javascript et les fichiers CSS des plugins à travers le Dispatcher est incroyablement inéfficace, il est chaudement recommandé de les symlinker pour la production. Par exemple comme ceci :

```
ln -s app/Plugin/YourPlugin/webroot/css/yourplugin.css app/webroot/css/yourplugin.css
```

CHAPITRE 12

Tutoriels et exemples

Dans cette section, vous pourrez découvrir des applications CakePHP typiques afin de voir comment toutes les pièces s'assemblent.

Sinon, vous pouvez vous référer au dépôt de plugins non-officiels de CakePHP CakePackages (http ://plugins.cakephp.org/) ainsi que la Boulangerie (http ://bakery.cakephp.org/) (Bakery) pour des applications et components existants.

Tutoriel d'un Blog

Bienvenue sur CakePHP. Vous consultez probablement ce tutoriel parce que vous voulez en apprendre plus à propos du fonctionnement de CakePHP. C'est notre but d'améliorer la productivité et de rendre le développement plus agréable : nous espérons que vous le découvrirez au fur et à mesure que vous plongerez dans le code.

Ce tutoriel vous accompagnera à travers la création d'une simple application de blog. Nous récupérerons et installerons Cake, créerons et configurerons une base de données et ajouterons suffisamment de logique applicative pour lister, ajouter, éditer et supprimer des posts.

Voici ce dont vous aurez besoin :

1. Un serveur web fonctionnel. Nous supposerons que vous utilisez Apache, bien que les instructions pour utiliser d'autres serveurs devraient être très semblables. Nous aurons peut-être besoin de jouer un peu sur la configuration du serveur, mais la plupart des personnes peuvent faire fonctionner Cake sans aucune configuration préalable.
2. Un serveur de base de données. Dans ce tutoriel, nous utiliserons MySQL. Vous aurez besoin d'un minimum de connaissance en SQL afin de créer une base de données : Cake prendra les rênes à partir de là.
3. Des connaissances de base en PHP. Plus vous aurez d'expérience en programmation orienté objet, mieux ce sera ; mais n'ayez crainte, même si vous êtes adepte de la programmation procédurale.
4. Enfin, vous aurez besoin de connaissances de base à propos du motif de conception MVC. Un bref aperçu de ce motif dans le chapitre *Comprendre le système M-V-C (Model-View-Controller)*. Ne vous inquiétez pas : il n'y a qu'une demi-page de lecture.

Maintenant, lançons-nous !

Obtenir Cake

Tout d'abord, récupérons une copie récente de Cake.

Pour obtenir la dernière version, allez sur le site GitHub du projet CakePHP : https ://github.com/cakephp/cakephp/tags et téléchargez la dernière version de la 2.0.

Vous pouvez aussi cloner le dépôt en utilisant git (http ://git-scm.com/). `git clone git://github.com/cakephp/cakephp.git`

Peu importe comment vous l'avez téléchargé, placez le code à l'intérieur du "DocumentRoot" de votre serveur. Une fois terminé, votre répertoire d'installation devrait ressembler à quelque chose comme cela :

```
/chemin_du_document_root
    /app
    /lib
    /plugins
    /vendors
    .htaccess
    index.php
    README
```

A présent, il est peut-être temps de voir un peu comment fonctionne la structure de fichiers de Cake : lisez le chapitre *Structure du dossier de CakePHP*.

Créer la base de données du blog

Maintenant, mettons en place la base de données pour notre blog. Si vous ne l'avez pas déjà fait, créez une base de données vide avec le nom de votre choix pour l'utiliser dans ce tutoriel. Pour le moment, nous allons juste créer une simple table pour stocker nos posts. Nous allons également insérer quelques posts à des fins de tests. Exécutez les requêtes SQL suivantes dans votre base de données :

```
/* D'abord, créons la table des posts : */
CREATE TABLE posts (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);

/* Puis insérons quelques posts pour les tests : */
INSERT INTO posts (title,body,created)
    VALUES ('Le titre', 'Voici le contenu du post.', NOW());
INSERT INTO posts (title,body,created)
    VALUES ('Encore un titre', 'Et le contenu du post qui suit.', NOW());
INSERT INTO posts (title,body,created)
    VALUES ('Le retour du titre', 'C\'est très excitant, non ?', NOW());
```

Le choix des noms pour les tables et les colonnes ne sont pas arbitraires. Si vous respectez les conventions de nommage Cake pour les bases de données et les classes (toutes deux expliquées le chapitre *Conventions de CakePHP*), vous tirerez profit d'un grand nombre de fonctionnalités automatiques et vous éviterez des étapes de configurations. Cake est suffisamment souple pour implémenter les pires schémas de bases de données, mais respecter les conventions vous fera gagner du temps.

Consultez le chapitre *Conventions de CakePHP* pour plus d'informations, mais il suffit de comprendre que nommer notre table 'posts' permet de la relier automatiquement à notre model Post, et qu'avoir des champs 'modified' et 'created' permet de les avoir gérés automagiquement par Cake.

Configurer la base de données Cake

En avant : indiquons à Cake où se trouve notre base de données et comment s'y connecter. Pour la plupart d'entre vous, c'est la première et dernière fois que vous configurerez quelque chose.

Une copie du fichier de configuration Cake pour la base de données se trouve dans `/app/Config/database.php.default`. Faites une copie de ce fichier dans le même répertoire mais nommez le `database.php`.

Le fichier de configuration devrait être assez simple : remplacez simplement les valeurs du tableau `$default` par celles qui correspondent à votre installation. Un exemple de tableau de configuration complet pourrait ressembler à ce qui suit :

```
public $default = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'port' => '',
    'login' => 'cakeBlog',
    'password' => 'c4k3-rUl3Z',
    'database' => 'cake_blog_tutorial',
    'schema' => '',
    'prefix' => '',
    'encoding' => ''
);
```

Une fois votre nouveau fichier `database.php` sauvegardé, vous devriez être en mesure d'ouvrir votre navigateur internet et de voir la page d'accueil de Cake. Elle devrait également vous indiquer votre fichier de connexion a été trouvé, et que Cake peut s'y connecter avec succès.

Note : Rappelez-vous que vous aurez besoin d'avoir PDO, et pdo_mysql activés dans votre php.ini.

Configuration facultative

Il y a trois autres élements qui peuvent être configurés. La plupart des développeurs configurent les éléments de cette petite liste, mais ils ne sont pas obligatoires pour ce tutoriel. Le premier consiste à définir une chaîne de caractères personnalisée (ou "grain de sel") afin de sécuriser les hashs. Le second consiste à définir un

nombre personnalisé (ou "graine") à utiliser pour le chiffrage. Le troisième est de permettre l'accès en écriture à CakePHP pour son dossier `tmp`.

Le "grain de sel" est utilisé pour générer des hashes. Changez sa valeur par défaut en modifiant `/app/Config/core.php` à la ligne 187. La nouvelle valeur n'a pas beaucoup d'importance du moment qu'elle est difficile à deviner :

```
/**
 * Une chaîne aléatoire utilisée dans les méthodes de hachage sécurisées.
 */
Configure::write('Security.salt', 'pl345e-P45s_7h3*S@l7!');
```

La "graine cipher" est utilisée pour le chiffrage/déchiffrage des chaînes de caractères. Changez sa valeur par défaut en modifiant `/app/Config/core.php` à la ligne 192. La nouvelle valeur n'a pas beaucoup d'importance du moment qu'elle est difficile à deviner :

```
/**
 * Une chaîne aléatoire de chiffre utilisée pour le chiffrage/déchiffrage
 * des chaînes de caractères.
 */
Configure::write('Security.cipherSeed', '74857126596251478436398467 51');
```

La dernière étape consiste à rendre le dossier `/app/tmp` accessible en écriture. Le meilleur moyen de faire cela est de trouver sous quel utilisateur votre serveur web s'exécute (`<?php echo `whoami`; ?>`) et de modifier le propriétaire du dossier `/app/tmp` pour cet utilisateur. La commande à exécuter (sous *nix) devrait ressembler à quelque chose comme cela

```
$ chown -R www-data app/tmp
```

Si pour une raison quelconque CakePHP ne peut pas écrire dans ce répertoire, vous en serez informé par un message d'avertissement tant que vous n'êtes pas en mode production.

Une note sur mod_rewrite

Occasionnellement, un nouvel utilisateur peut avoir des problèmes de mod_rewrite. par exemple si la page d'accueil de CakePHP a l'air marrante (pas d'images ou de styles css), cela signifie probablement que mod_rewrite ne fonctionne pas sur votre système. Merci de vous référer à la section suivante sur l'url rewriting pour que votre serveur web fonctionne :

URL Rewriting

Apache et mod_rewrite (et .htaccess)

Alors que CakePHP est construit pour travailler avec mod_rewrite –et habituellement il l'est– nous avons remarqué que certains utilisateurs se battent pour obtenir un bon fonctionnement sur leurs systèmes.

Ici il y a quelques trucs que vous pourriez essayer pour que cela fonctionne correctement. Premièrement, regardez votre fichier httpd.conf (Assurez vous que vous avez éditer le httpd.conf du système plutôt que celui d'un utilisateur- ou le httpd.conf d'un site spécifique).

1. Assurez-vous qu'un .htaccess est permis et que AllowOverride est défini à All pour le DocumentRoot correct. Vous devriez voir quelque chose comme :

```
# Chaque répertoire auquel Apache a accès peut être configuré avec
# respect pour lesquels les services et les fonctionnalités sont
# autorisés et/ou désactivés dans ce répertoire (et ses sous-répertoires).
#
# Premièrement, nous configurons "par défault" pour être un ensemble
# très restrictif de fonctionnalités.
#
<Directory />
    Options FollowSymLinks
    AllowOverride All
#    Order deny,allow
#    Deny from all
</Directory>
```

2. Assurez-vous que vous avez chargé correctement mod_rewrite. Vous devriez voir quelque chose comme :

```
LoadModule rewrite_module libexec/apache2/mod_rewrite.so
```

   Dans la plupart des systèmes, ceux-ci vont être commentés (en étant précédé par un #) par défaut, donc vous aurez juste besoin de retirer les symboles # du début.

   Après que vous avez fait des changements, re-démarrez Apache pour être sûr que les paramètres soient actifs.

   Vérifiez que vos fichiers .htaccess sont effectivement dans le bon répertoire.

   Cela peut arriver pendant la copie parce que certains systèmes d'exploitation traitent les fichiers qui commencent par '.' en caché et du coup ne les voient pas pour les copier.

3. Assurez-vous que votre copie de CakePHP vient de la section des téléchargements du site de notre dépôt GIT, et a été dézippé correctement en vérifiant les fichiers .htaccess.

   Le répertoire root de Cake (a besoin d'être copié dans votre document, cela redirige tout vers votre app Cake) :

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$ app/webroot/    [L]
   RewriteRule    (.*) app/webroot/$1 [L]
</IfModule>
```

   Le répertoire app de Cake (sera copié dans le répertoire supérieur de votre application avec Bake) :

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$    webroot/    [L]
   RewriteRule    (.*) webroot/$1    [L]
</IfModule>
```

   Le répertoire webroot de Cake (sera copié dans le webroot de votre application avec Bake) :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php/$1 [QSA,L]
</IfModule>
```

Si votre site Cakephp a toujours des problèmes avec mod_rewrite, essayez de modifier les paramètres pour les virtualhosts. Si vous êtes sur ubuntu, modifiez le fichier /etc/apache2/sites-available/default (l'endroit dépend de la distribution). Dans ce fichier, assurez-vous que `AllowOverride None` a changé en `AllowOverride All`, donc vous avez :

```
<Directory />
    Options FollowSymLinks
    AllowOverride All
</Directory>
<Directory /var/www>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order Allow,Deny
    Allow from all
</Directory>
```

Si vous êtes sur Mac OSX, une autre solution est d'utiliser l'outil virtualhostx pour faire un hôte virtuel pour pointer vers votre dossier.

Pour beaucoup de services d'hébergement (GoDaddy, 1and1), votre serveur web est en fait déjà distribué à partir d'un répertoire utilisateur qui utilise déjà mod_rewrite. Si vous installez CakePHP dans un répertoire utilisateur (http ://exemple.com/~username/cakephp/), ou toute autre structure d'URL qui utilise déjà mod_rewrite, vous aurez besoin d'ajouter les requêtes (statements) RewriteBase aux fichiers .htaccess que CakePHP utilise (/.htaccess, /app/.htaccess, /app/webroot/.htaccess).

Ceci peut être ajouté à la même section avec la directive RewriteEngine, donc par exemple, votre fichier .htaccess dans webroot ressemblerait à ceci :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/cake/app
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php/$1 [QSA,L]
</IfModule>
```

Les détails de ces changements dépendront de votre configuration, et pourront inclure des choses supplémentaires qui ne sont pas liées à CakePHP. Merci de vous renseigner sur la documentation en ligne d'Apache pour plus d'informations.

4. (Optionel) Pour améliorer la configuration de production, vous devriez empêcher les assets invalides d'être parsés par CakePHP. Modifier votre webroot .htaccess pour quelque chose comme :

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/cake/app
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
```

```
        RewriteCond %{REQUEST_URI} !^/(app/webroot/)?(img|css|js)/(.*)$
        RewriteRule ^(.*)$ index.php [QSA,L]
    </IfModule>
```

Ce qui est au-dessus va simplement empêcher les assets incorrects d'être envoyés à index.php et à la place d'afficher la page 404 de votre serveur web.

De plus, vous pouvez créer une page HTML 404 correspondante, ou utiliser la page 404 de CakePHP intégrée en ajoutant une directive `ErrorDocument` :

```
    ErrorDocument 404 /404-not-found
```

De belles URLs sur nginx

nginx est un serveur populaire qui, comme Lighttpd, utilise moins de ressources système. Son inconvénient est qu'il ne fait pas usage de fichiers .htaccess comme Apache et Lighttpd, il est donc nécessaire de créer les URLs réécrites dans la configuration du site disponibles. selon votre configuration, vous devrez modifier cela, mais à tout le moins, vous aurez besoin de PHP fonctionnant comme une instance FastCGI.

```
server {
    listen   80;
    server_name www.example.com;
    rewrite ^(.*) http://example.com$1 permanent;
}

server {
    listen   80;
    server_name example.com;

    # root directive should be global
    root   /var/www/example.com/public/app/webroot/;
    index  index.php;

    access_log /var/www/example.com/log/access.log;
    error_log /var/www/example.com/log/error.log;

    location / {
        try_files $uri $uri/ /index.php?$uri&$args;
    }

    location ~ \.php$ {
        include /etc/nginx/fastcgi_params;
        try_files $uri =404;
        fastcgi_pass    127.0.0.1:9000;
        fastcgi_index   index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

Rewrites d'URL sur IIS7 (serveurs Windows)

IIS7 ne supporte pas nativement les fichiers .htaccess. Bien qu'il existe des add-ons qui peuvent ajouter ce support, vous pouvez aussi importer les règles des .htaccess dans IIS pour utiliser les rewrites natifs de CakePHP. Pour ce faire, suivez ces étapes :

1. Utilisez l'installeur de la plateforme Web de Microsoft (http ://www.microsoft.com/web/downloads/platform.aspx) pour installer l'URL Rewrite Module 2.0 (http ://www.iis.net/downloads/microsoft/url-rewrite) ou télécharger le directement (32-bit (http ://www.microsoft.com/en-us/download/details.aspx ?id=5747) / 64-bit (http ://www.microsoft.com/en-us/download/details.aspx ?id=7435)).
2. Créez un nouveau fichier dans votre dossier CakePHP, appelé web.config.
3. Utilisez Notepad ou tout autre éditeur XML-safe, copiez le code suivant dans votre nouveau fichier web.config...

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="Rewrite requests to test.php" stopProcessing="true">
                    <match url="^test.php(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="app/webroot/test.php{R:1}" />
                </rule>
                <rule name="Exclude direct access to app/webroot/*" stopProcessing="true">
                    <match url="^app/webroot/(.*)$" ignoreCase="false" />
                    <action type="None" />
                </rule>
                <rule name="Rewrite routed access to assets (img, css, files, js, favicon)'
                    <match url="^(img|css|files|js|favicon.ico)(.*)$" />
                    <action type="Rewrite" url="app/webroot/{R:1}{R:2}" appendQueryString='
                </rule>
                <rule name="Rewrite requested file/folder to index.php" stopProcessing="tru
                    <match url="^(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="index.php" appendQueryString="true" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

Une fois que le fichier web.config est créé avec les bonnes règles de réécriture des liens de IIS, les liens CakePHP, les CSS, les JS, et le reroutage devraient fonctionner correctement.

Je ne veux / ne peux utiliser l'URL rewriting

Si vous ne voulez ou ne pouvez pas utiliser l'URL rewriting sur votre serveur web, référez-vous à la section *core configuration*.

Maintenant continuez sur *Blog Tutoriel - Ajouter la logique* pour commencer à construire votre première application CakePHP.

Blog Tutoriel - Ajouter la logique

Créer un model Post

La classe Model est le pain quotidien des applications CakePHP. En créant un model CakePHP qui interagira avec notre base de données, nous aurons mis en place les fondations nécessaires pour faire plus tard nos opérations de lecture, d'insertion, d'édition et de suppression.

Les fichiers des classes Model de CakePHP se trouvent dans `/app/Model`, et le fichier que nous allons créer maintenant sera enregistré dans `/app/Model/Post.php`. Le fichier complet devrait ressembler à ceci

```
class Post extends AppModel {
}
```

La convention de nommage est vraiment très importante dans CakePHP. En nommant notre model Post, CakePHP peut automatiquement déduire que ce model sera utilisé dans le controller PostsController, et sera lié à la table `posts` de la base de données.

Note : CakePHP créera dynamiquement un objet model pour vous, s'il ne trouve pas le fichier correspondant dans /app/Model. Cela veut aussi dire que si vous n'avez pas nommé correctement votre fichier (par ex. post.php ou posts.php). CakePHP ne reconnaîtra pas votre configuration et utilisera ses objets model par défaut.

Pour plus d'informations sur les models, comme les préfixes des tables, les callbacks, et la validation, consultez le chapitre *Models (Modèles)* du manuel.

Créer un controller Posts

Nous allons maintenant créer un controller pour nos posts. Le controller est l'endroit où s'exécutera toute la logique métier pour l'intéraction du processus de post. En un mot, c'est l'endroit où vous jouerez avec les models et où les tâches liées aux posts s'exécutent. Nous placerons ce nouveau controller dans un fichier appelé `PostsController.php` à l'intérieur du dossier `/app/Controller`. Voici à quoi devrait ressembler le controller de base

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');
}
```

Maintenant, ajoutons une action à notre controller. Les actions représentent souvent une simple fonction ou une interface dans une application. Par exemple, quand les utilisateurs requêtent www.exemple.com/posts/index (qui est également la même chose que www.exemple.com/posts/), ils pourraient s'attendre à voir une liste de posts. Le code pour cette action devrait ressembler à quelque chose comme ça :

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');
```

```
    public function index() {
        $this->set('posts', $this->Post->find('all'));
    }
}
```

Laissez-moi vous expliquer un peu cette action. En définissant la fonction `index()` dans notre PostsController, les utilisateurs peuvent maintenant accéder à cette logique en demandant www.exemple.com/posts/index. De la même façon, si nous devions définir une fonction appelée `foobar()`, les utilisateurs pourrait y accéder en demandant www.exemple.com/posts/foobar.

Warning : Vous pourriez être tenté de nommer vos controllers et vos actions d'une certaine manière pour obtenir une certaine URL. Résistez à cette tentation. Suivez les conventions CakePHP (le nom des controllers au pluriel, etc.) et nommez vos actions de façon lisible et compréhensible. Vous pouvez lier les URLs à votre code en utilisant ce qu'on appelle des "routes", on le verra plus tard.

La seule instruction que cette action utilise est `set()`, pour transmettre les données du controller à la vue (que nous créerons à la prochaine étape). La ligne définit la variable de vue appelée 'posts' qui est égale à la valeur de retour de la méthode `find('all')` du model Post. Notre model Post est automatiquement disponible via $this->Post, parce que nous avons suivi les conventions de nommage de Cake.

Pour en apprendre plus sur les controllers de Cake, consultez le chapitre *Controllers (Contrôleurs)*.

Créer les Vues des Posts

Maintenant que nous avons nos données en provenance du model, ainsi que la logique applicative et les flux définis par notre controller, nous allons créer une vue pour l'action "index" que nous avons créé ci-dessus.

Les vues de Cake sont juste des fragments de présentation "assaisonnée", qui s'intègrent au sein d'un layout applicatif. Pour la plupart des applications, elles sont un mélange de HTML et PHP, mais les vues peuvent aussi être constituées de XML, CSV ou même de données binaires.

Les Layouts sont du code de présentation, encapsulé autour d'une vue, ils peuvent être définis et interchangés, mais pour le moment, utilisons juste celui par défaut.

Vous souvenez-vous, dans la dernière section, comment nous avions assigné la variable 'posts' à la vue en utilisant la méthode `set()` ? Cela devrait transmettre les données à la vue qui ressemblerait à quelque chose comme ça :

```
// print_r($posts) sort:

Array
(
    [0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => Le titre
                    [body] => Voici le contenu du post.
                    [created] => 2008-02-13 18:34:55
                    [modified] =>
```

```
                )
        )
    [1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => Encore un titre
                    [body] => Et le contenu du post qui suit.
                    [created] => 2008-02-13 18:34:56
                    [modified] =>
                )
        )
    [2] => Array
        (
            [Post] => Array
                (
                    [id] => 3
                    [title] => Le retour du titre
                    [body] => C'est très excitant, non ?
                    [created] => 2008-02-13 18:34:57
                    [modified] =>
                )
        )
)
```

Les fichiers des vues de Cake sont stockés dans `/app/views` à l'intérieur d'un dossier dont le nom correspond à celui du controller (nous aurons à créer un dossier appelé 'posts' dans ce cas). Pour mettre en forme les données de ces posts dans un joli tableau, le code de notre vue devrait ressembler à quelque chose comme cela :

```
<!-- File: /app/View/Posts/index.ctp -->

<h1>Blog posts</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
    </tr>

    <!-- Here is where we loop through our $posts array, printing out post info -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'],
            array('controller' => 'posts', 'action' => 'view', $post['Post']['id'])); ?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
    <?php endforeach; ?>
```

```
    <?php unset($post); ?>
</table>
```

Bien entendu, cela donnera quelque chose de simple.

Vous avez sans doute remarqué l'utilisation d'un objet appelé `$this->Html`. C'est une instance de la classe CakePHP `HtmlHelper`. CakePHP est livré avec un ensemble de "helpers" (des assistants) pour les vues, qui réalisent en un clin d'oeil des choses comme le "linking" (mettre les liens dans un texte), l'affichage des formulaires, du JavaScript et de l'Ajax. Vous pouvez en apprendre plus sur la manière de les utiliser dans le chapitre *Helpers (Assistants)*, mais ce qu'il est important de noter ici, c'est que la méthode `link()` génèrera un lien HTML à partir d'un titre (le premier paramètre) et d'une URL (le second paramètre).

Lorsque vous indiquez des URLs dans Cake, il est recommandé d'utiliser les tableaux. Ceci est expliqué dans le chapitre des Routes. Utilisez les tableaux dans les URLs, vous permet de tirer profit des capacités de CakePHP à ré-inverser les routes. Vous pouvez aussi utiliser les URLs relatives depuis la base de l'application comme suit /controller/action/param1/param2.

A ce stade, vous devriez être en mesure de pointer votre navigateur sur la page http ://www.exemple.com/posts/index. Vous devriez voir votre vue, correctement formatée avec le titre et le tableau listant les posts.

Si vous avez essayé de cliquer sur l'un des liens que nous avons créés dans cette vue (le lien sur le titre d'un post mène à l'URL : /posts/view/un_id_quelconque), vous avez sûrement été informé par CakePHP que l'action n'a pas encore été définie. Si vous n'avez pas été informé, soit quelque chose s'est mal passé, soit en fait vous aviez déjà défini l'action, auquel cas vous êtes vraiment sournois ! Sinon, nous allons la créer sans plus tarder dans le Controller Posts

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');

    public function index() {
         $this->set('posts', $this->Post->find('all'));
    }

    public function view($id = null) {
        if (!$id) {
            throw new NotFoundException(__('Invalid post'));
        }

        $post = $this->Post->findById($id);
        if (!$post) {
            throw new NotFoundException(__('Invalid post'));
        }
        $this->set('post', $post);
    }
}
```

L'appel de `set()` devrait vous être familier. Notez que nous utilisons `read()` plutôt que `find('all')` parce que nous voulons seulement récupérer les informations d'un seul post.

Notez que notre action "view" prend un paramètre : l'ID du post que nous aimerions voir. Ce paramètre est transmis à l'action grâce à l'URL demandée. Si un utilisateur demande /posts/view/3, alors la valeur '3' est transmise à la variable `$id`.

Nous faisons aussi une petite vérification d'erreurs pour nous assurer qu'un utilisateur accède bien à l'enregsitrement. Si un utilisateur requête `/posts/view`, nous lancerons un `NotFoundException` et laisserons le Gestionnaire d'Erreur de CakePHP ErrorHandler prendre le dessus. Nous executons aussi une vérification similaire pour nous assurer que l'utilisateur a accède à un enregistrement qui existe.

Maintenant, créons la vue pour notre nouvelle action "view" et plaçons-la dans `/app/View/Posts/view.ctp`.

```
<!-- Fichier : /app/View/Posts/view.ctp -->

<h1><?php echo h($post['Post']['title']); ?></h1>

<p><small>Créé le : <?php echo $post['Post']['created']; ?></small></p>

<p><?php echo h($post['Post']['body']); ?></p>
```

Vérifiez que cela fonctionne en testant les liens de la page /posts/index ou en affichant manuellement un post via `/posts/view/1`.

Ajouter des Posts

Lire depuis la base de données et nous afficher les posts est un bon début, mais lançons-nous dans l'ajout de nouveaux posts.

Premièrement, commençons par créer une action `add()` dans le PostsController :

```
class PostsController extends AppController {
    public $helpers = array('Html', 'Form');
    public $components = array('Session');

    public function index() {
        $this->set('posts', $this->Post->find('all'));
    }

    public function view($id) {
        if (!$id) {
            throw new NotFoundException(__('Invalid post'));
        }

        $post = $this->Post->findById($id);
        if (!$post) {
            throw new NotFoundException(__('Invalid post'));
        }
        $this->set('post', $post);
    }

    public function add() {
        if ($this->request->is('post')) {
            $this->Post->create();
            if ($this->Post->save($this->request->data)) {
                $this->Session->setFlash(__('Votre post a été sauvegardé.'));
                $this->redirect(array('action' => 'index'));
```

```
            } else {
                $this->Session->setFlash(__('Impossible d\'ajouter votre post.'));
            }
        }
    }
}
```

Note : Vous avez besoin d'inclure le component Session (SessionComponent) et le helper Session (SessionHelper) dans chaque controller que vous utiliserez. Si nécessaire, incluez-les dans le controller principal (AppController) pour qu'ils soient accessibles à tous les controllers.

Voici ce que fait l'action `add()` : si la requête HTTP est de type POST, essayez de sauvegarder les données en utilisant le model "Post". Si pour une raison quelconque, la sauvegarde a échouée, affichez simplement la vue. Cela nous donne une chance de voir les erreurs de validation de l'utilisateur et d'autres avertissements.

Chaque requête de CakePHP contient un objet `CakeRequest` qui est accessible en utilisant `$this->request`. Cet objet contient des informations utiles sur la requête qui vient d'être reçue, et permet de contrôler les flux de votre application. Dans ce cas, nous utilisons la méthode `CakeRequest::is()` ` pour vérifier que la requête est de type POST.

Lorsqu'un utilisateur utilise un formulaire pour poster des données dans votre application, ces informations sont disponibles dans `$this->request->data`. Vous pouvez utiliser les fonctions `pr()` ou `debug()` pour les afficher si vous voulez voir à quoi cela ressemble.

Nous utilisons la méthode `SessionComponent::setFlash()` du component Session (SessionComponent) pour définir un message dans une variable session et qui sera affiché dans la page juste après la redirection. Dans le layout, nous trouvons la fonction `SessionHelper::flash` qui permet d'afficher et de nettoyer la variable correspondante. La méthode `Controller::redirect` ` du controller permet de rediriger vers une autre URL. Le paramètre `array('action' => 'index')` sera traduit vers l'URL /posts, c'est à dire l'action "index" du controller "Posts" (PostsController). Vous pouvez vous référer à l'API de la fonction `Router::url()` ` pour voir les différents formats d'URL acceptés dans les différentes fonctions de Cake.

L'appel de la méthode `save()` vérifiera les erreurs de validation et interrompra l'enregistrement s'il y en a une qui survient. Nous verrons la façon dont les erreurs sont traitées dans les sections suivantes.

Valider les données

Cake place la barre très haute pour briser la monotonie de la validation des champs de formulaires. Tout le monde déteste le dévelopement de formulaires interminables et leurs routines de validations. Cake rend tout cela plus facile et plus rapide.

Pour tirer profit des fonctionnalités de validation, vous devez utiliser le helper "Form" (FormHelper) dans vos vues. `FormHelper` est disponible par défaut dans toutes les vues avec la variables `$this->Form`.

Voici le code de notre vue "add" (ajout)

```
<!-- Fichier : /app/View/Posts/add.ctp -->

<h1>Ajouter un post</h1>
```

```
<?php
echo $this->Form->create('Post');
echo $this->Form->input('title');
echo $this->Form->input('body', array('rows' => '3'));
echo $this->Form->end('Sauvegarder le post');
?>
```

Nous utilisons ici le `FormHelper` pour générer la balise d'ouverture d'une formulaire HTML. Voici le code HTML généré par `$this->Form->create()` :

```
.. code-block:: html
```

<form id="PostAddForm" method="post" action="/posts/add">

Si `create()` est appelée sans aucun paramètre, Cake suppose que vous construisez un formulaire qui envoie les données en POST à l'action `add()` (ou `edit()` quand `id` est dans les données du formulaire) du controller actuel.

La méthode `$this->Form->input()` est utilisée pour créer des élements de formulaire du même nom. Le premier paramètre dit à CakePHP à quels champs ils correspondent et le second paramètre vous permet de spécifier un large éventail d'options - dans ce cas, le nombre de lignes du textarea. Il y a un peu d'introspection et "d'automagie" ici : `input()` affichera différents éléments de formulaire selon le champ spécifié du model.

L'appel de la méthode `$this->Form->end()` génère un bouton de soumission et ajoute la balise de fermeture du formulaire. Si une chaîne de caractères est passée comme premier paramètre de la méthode `end()`, le helper "Form" affichera un bouton de soumission dont le nom correspond à celle-ci. Encore une fois, référez-vous au chapitre *Helpers (Assistants)* pour en savoir plus sur les helpers.

A présent, revenons en arrière et modifions notre vue `/app/View/Posts/index.ctp` pour ajouter un lien "Ajouter un post". Ajoutez la ligne suivante avant `<table>`

```
<?php echo $this->Html->link(
    'Ajouter un Post',
    array('controller' => 'posts', 'action' => 'add')
); ?>
```

Vous vous demandez peut-être : comment je fais pour indiquer à CakePHP mes exigences de validation ? Les règles de validation sont définies dans le model. Retournons donc à notre model Post et précédons à quelques ajustements

```
class Post extends AppModel {
    public $validate = array(
        'title' => array(
            'rule' => 'notEmpty'
        ),
        'body' => array(
            'rule' => 'notEmpty'
        )
    );
}
```

Le tableau `$validate` indique à CakePHP comment valider vos données lorsque la méthode `save()` est appelée. Ici, j'ai spécifié que les deux champs "body" et "title" ne doivent pas être vides. Le moteur de validation de CakePHP est puissant, il dispose d'un certain nombre de règles pré-fabriquées (code de carte bancaire, adresse emails, etc.) et d'une souplesse pour ajouter vos propres règles de validation. Pour plus d'informations sur cette configuration, consultez le chapitre *Validation des Donnnées*.

Maintenant que vos règles de validation sont en place, utilisez l'application pour essayer d'ajouter un post avec un titre et un contenu vide afin de voir comment cela fonctionne. Puisque que nous avons utilisé la méthode `FormHelper::input()` ` du helper "Form" pour créer nos éléments de formulaire, nos messages d'erreurs de validation seront affichés automatiquement.

Editer des Posts

L'édition de posts : nous y voilà. Vous êtes un pro de CakePHP maintenant, vous devriez donc avoir adopté le principe. Créez d'abord l'action puis la vue. Voici à quoi l'action `edit()` du controller Posts (PostsController) devrait ressembler

```
public function edit($id = null) {
    if (!$id) {
        throw new NotFoundException(__('Invalid post'));
    }

    $post = $this->Post->findById($id);
    if (!$post) {
        throw new NotFoundException(__('Invalid post'));
    }

    if ($this->request->is('post') || $this->request->is('put')) {
        $this->Post->id = $id;
        if ($this->Post->save($this->request->data)) {
            $this->Session->setFlash(__('Your post has been updated.'));
            $this->redirect(array('action' => 'index'));
        } else {
            $this->Session->setFlash(__('Unable to update your post.'));
        }
    }

    if (!$this->request->data) {
        $this->request->data = $post;
    }
}
```

Cette action s'assure d'abord que l'utilisateur a essayé d'accéder à un enregistrement existant. Si il n'y a pas de paramètre `$id`` `passé, ou si le post n'existe pas, nous lançons une ``NotFoundException` pour que le gestionnaire d'Erreurs ErrorHandler de CakePHP s'en occupe.

Ensuite l'action vérifie que la requête est une requête POST. Si elle l'est, alors nous utilisons les données POST pour mettre à jour notre enregistrement Post, ou sortir et montrer les erreurs de validation à l'utilisateur.

Si il n'y a pas de données définies dans `$this->request->data`, nous le définissons simplement dans le post récupéré précédemment.

La vue d'édition devrait ressembler à quelque chose comme cela :

```
<!-- Fichier: /app/View/Posts/edit.ctp -->

<h1>Editer le post</h1>
<?php
    echo $this->Form->create('Post', array('action' => 'edit'));
    echo $this->Form->input('title');
    echo $this->Form->input('body', array('rows' => '3'));
    echo $this->Form->input('id', array('type' => 'hidden'));
    echo $this->Form->end('Sauvegarder le post');
```

Cette vue affiche le formulaire d'édition (avec les données pré-remplies) avec les messages d'erreur de validation nécessaires.

Une chose à noter ici : CakePHP supposera que vous éditez un model si le champ 'id' est présent dans le tableau de données. S'il n'est pas présent (ce qui revient à notre vue "add"), Cake supposera que vous insérez un nouveau model lorsque `save()` sera appelée.

Vous pouvez maintenant mettre à jour votre vue "index" avec des liens pour éditer des posts :

```
<!-- Fichier: /app/View/Posts/index.ctp  (lien d\'édition ajouté) -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link("Ajouter un Post", array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Titre</th>
        <th>Action</th>
        <th>Créé le</th>
    </tr>

<!-- Ici se trouve la boucle de notre tableau $posts, impression de l\'info du post -->

<?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'], array('action' => 'view',
        </td>
        <td>
            <?php echo $this->Html->link('Edit', array('action' => 'edit', $post['Post']['
        </td>
        <td>
            <?php echo $post['Post']['created']; ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Supprimer des Posts

A présent, mettons en place un moyen de supprimer les posts pour les utilisateurs. Démarrons avec une action `delete()` dans le controller Posts (PostsController) :

```
public function delete($id) {
    if ($this->request->is('get')) {
        throw new MethodNotAllowedException();
    }
    if ($this->Post->delete($id)) {
        $this->Session->setFlash(__('Le Post avec l\'id ' . $id . ' a été supprimé.'));
        $this->redirect(array('action' => 'index'));
    }
}
```

Cette logique supprime le Post spécifié par $id, et utilise `$this->Session->setFlash()` pour afficher à l'utilisateur un message de confirmation après l'avoir redirigé sur `/posts`. Si l'utilisateur tente une suppression en utilisant une requête GET, une exeception est levée. Les exceptions manquées sont capturées par le gestionnaire d'exceptions de CakePHP et un joli message d'erreur est affiché. Il y a plusieurs *Exceptions* intégrées qui peuvent être utilisées pour indiquer les différentes erreurs HTTP que votre application pourrait rencontrer.

Etant donné que nous exécutons juste un peu de logique et de redirection, cette action n'a pas de vue. Vous voudrez peut-être mettre à jour votre vue "index" avec des liens pour permettre aux utilisateurs de supprimer des Posts, ainsi :

```
<!-- Fichier: /app/View/Posts/index.ctp -->

<h1>Blog posts</h1>
<p><?php echo $this->Html->link('Ajouter un Post', array('action' => 'add')); ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Titre</th>
        <th>Actions</th>
        <th>Créé le</th>
    </tr>

<!-- Ici, nous bouclons sur le tableau $post afin d'afficher les informations des posts -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $this->Html->link($post['Post']['title'], array('action' => 'view',
        </td>
        <td>
            <?php echo $this->Form->postLink(
                'Delete',
                array('action' => 'delete', $post['Post']['id']),
                array('confirm' => 'Etes-vous sûr ?'));
            ?>
            <?php echo $this->Html->link('Editer', array('action' => 'edit', $post['Post']
```

```
        </td>
        <td>
            <?php echo $post['Post']['created']; ?>
        </td>
    </tr>
    <?php endforeach; ?>

</table>
```

Utiliser `postLink()` permet de créer un lien qui utilise du Javascript pour supprimer notre post en faisant une requête POST. Autoriser la suppression par une requête GET est dangereux à cause des robots d'indexation qui peuvent tous les supprimer.

Note : Ce code utilise aussi le helper "Form" pour demander à l'utilisateur une confirmation avant de supprimer le post.

Routes

Pour certains, le routage par défaut de CakePHP fonctionne suffisamment bien. Les développeurs qui sont sensibles à la facilité d'utilisation et à la compatibilité avec les moteurs de recherches apprécieront la manière dont CakePHP lie des URLs à des actions spécifiques. Nous allons donc faire une rapide modification des routes dans ce tutoriel.

Pour plus d'informations sur les techniques de routages, consultez le chapitre *Configuration des Routes*.

Par défaut, CakePHP effectue une redirection d'une personne visitant la racine de votre site (i.e. http ://www.exemple.com) vers le controller Pages (PagesController) et affiche le rendu de la vue appelée "home". Au lieu de cela, nous voudrions la remplacer avec notre controller Posts (PostsController).

Le routage de Cake se trouve dans `/app/Config/routes.php`. Vous devrez commenter ou supprimer la ligne qui définit la route par défaut. Elle ressemble à cela

```
Router::connect('/', array('controller' => 'pages', 'action' => 'display', 'home'));
```

Cette ligne connecte l'URL '/' à la page d'accueil par défaut de CakePHP. Nous voulons que cette URL soit connectée à notre propre controller, remplacez donc la ligne par celle-ci

```
Router::connect('/', array('controller' => 'posts', 'action' => 'index'));
```

Cela devrait connecter les utilisateurs demandant '/' à l'action `index()` de notre controller Posts (PostsController).

Note : CakePHP peut aussi faire du 'reverse routing' (ou routage inversé). Par exemple, pour la route définie plus haut, en ajoutant `array('controller' => 'posts', 'action' => 'index')` à la fonction retournant un tableau, l'URL '/' sera utilisée. Il est d'ailleurs bien avisé de toujours utiliser un tableau pour les URLs afin que vos routes définissent où vont les URLs, mais aussi pour s'assurer qu'elles aillent dans la même direction.

Conclusion

Créer des applications de cette manière vous apportera, paix, honneur, amour et argent au-delà même de vos fantasmes les plus fous. Simple n'est ce pas ? Gardez à l'esprit que ce tutoriel était très basique. CakePHP a *beaucoup* plus de fonctionnalités à offrir et il est aussi souple dans d'autres domaines que nous n'avons pas souhaité couvrir ici pour simplifier les choses. Utilisez le reste de ce manuel comme un guide pour développer des applications plus riches en fonctionnalités.

Maintenant que vous avez créé une application Cake basique, vous êtes prêt pour les choses sérieuses. Commencez votre propre projet et lisez le reste du Manuel et de l'API (http ://api20.cakephp.org).

Si vous avez besoin d'aide, venez nous voir sur le canal IRC #cakephp. Bienvenue sur CakePHP !

Prochaines lectures suggérées

Voici les différents chapitres que les gens veulent souvent lire après :

1. *Layouts* : Personnaliser les Layouts de votre application.
2. *Elements* : Inclure et ré-utiliser les portions de vues.
3. *Scaffolding* : Construire une ébauche d'application sans avoir à coder.
4. *Génération de code avec Bake* Générer un code CRUD basique.
5. *Authentification Simple et Autorisation de l'Application* : Tutoriel sur l'enregistrement et la connexion d'utilisateurs.

Authentification Simple et Autorisation de l'Application

Suivez notre exemple *Tutoriel d'un Blog*, imaginons que nous souhaitions sécuriser l'accès de certaines urls, basées sur la connexion de l'user. Nous avons aussi une autre condition requise, qui est d'autoriser notre blog à avoir des auteurs multiples, afin que chacun d'eux puisse créer ses propres posts, les modifier et les supprimer selon le besoin interdisant d'autres auteurs à apporter des modifications sur ses messages.

Créer le code lié de tous les users

Premièrement, créeons une nouvelle table dans notre base de données du blog pour contenir les données de notre user :

```
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50),
    password VARCHAR(50),
    role VARCHAR(20),
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

Nous avons respecté les conventions de CakePHP pour le nommage des tables, mais nous profitons d'une autre convention : en utilisant les colonnes du nom d'user et du mot de passe dans une table users, CakePHP sera capable de configurer automatiquement la plupart des choses pour nous quand on réalisera la connexion de l'user.

La prochaine étape est de créer notre model User, qui a la responsablilité de trouver, sauvegarder et valider toute donnée d'user :

```
// app/Model/User.php
class User extends AppModel {
    public $name = 'User';
    public $validate = array(
        'username' => array(
            'required' => array(
                'rule' => array('notEmpty'),
                'message' => 'Un nom d\'user est requis'
            )
        ),
        'password' => array(
            'required' => array(
                'rule' => array('notEmpty'),
                'message' => 'Un mot de passe est requis'
            )
        ),
        'role' => array(
            'valid' => array(
                'rule' => array('inList', array('admin', 'auteur')),
                'message' => 'Merci de rentrer un rôle valide',
                'allowEmpty' => false
            )
        )
    );
}
```

Créeons aussi notre UsersController, le contenu suivant correspond à la classe *cuisinée* basique UsersController en utilisant les utilitaires de génération de code fournis avec CakePHP :

```
// app/Controller/UsersController.php
class UsersController extends AppController {

    public function beforeFilter() {
        parent::beforeFilter();
        $this->Auth->allow('add', 'logout');
    }

    public function index() {
        $this->User->recursive = 0;
        $this->set('users', $this->paginate());
    }

    public function view($id = null) {
        $this->User->id = $id;
        if (!$this->User->exists()) {
```

```
            throw new NotFoundException(__('User invalide'));
        }
        $this->set('user', $this->User->read(null, $id));
    }

    public function add() {
        if ($this->request->is('post')) {
            $this->User->create();
            if ($this->User->save($this->request->data)) {
                $this->Session->setFlash(__('L\'user a été sauvegardé'));
                $this->redirect(array('action' => 'index'));
            } else {
                $this->Session->setFlash(__('L\'user n\'a pas été sauvegardé. Merci de rées
            }
        }
    }

    public function edit($id = null) {
        $this->User->id = $id;
        if (!$this->User->exists()) {
            throw new NotFoundException(__('User Invalide'));
        }
        if ($this->request->is('post') || $this->request->is('put')) {
            if ($this->User->save($this->request->data)) {
                $this->Session->setFlash(__('L\'user a été sauvegardé'));
                $this->redirect(array('action' => 'index'));
            } else {
                $this->Session->setFlash(__('L\'user n\'a pas été sauvegardé. Merci de rées
            }
        } else {
            $this->request->data = $this->User->read(null, $id);
            unset($this->request->data['User']['password']);
        }
    }

    public function delete($id = null) {
        if (!$this->request->is('post')) {
            throw new MethodNotAllowedException();
        }
        $this->User->id = $id;
        if (!$this->User->exists()) {
            throw new NotFoundException(__('User invalide'));
        }
        if ($this->User->delete()) {
            $this->Session->setFlash(__('User supprimé'));
            $this->redirect(array('action' => 'index'));
        }
        $this->Session->setFlash(__('L\'user n\'a pas été supprimé'));
        $this->redirect(array('action' => 'index'));
    }
```

De la même façon, nous avons crée les vues pour nos posts de blog ou en utilisant l'outil de génération de code, nous exécutons les vues. Dans le cadre de ce tutoriel, nous allons juste montrer le add.ctp :

```
<!-- app/View/Users/add.ctp -->
<div class="users form">
<?php echo $this->Form->create('User');?>
    <fieldset>
        <legend><?php echo __('Ajouter User'); ?></legend>
        <?php echo $this->Form->input('username');
        echo $this->Form->input('password');
        echo $this->Form->input('role', array(
            'options' => array('admin' => 'Admin', 'auteur' => 'Auteur')
        ));
    ?>
    </fieldset>
<?php echo $this->Form->end(__('Ajouter'));?>
</div>
```

Authentification (Connexion et Deconnexion)

Nous sommes maintenant prêt à ajouter notre couche d'authentification. Dans CakePHP, c'est géré par `AuthComponent`, une classe responsable d'exiger la connexion pour certaines actions, de gérer la connexion et la déconnexion, et aussi d'autoriser aux users connectés les actions que l'on souhaite leur voir autorisées.

Pour ajouter ce component à votre application, ouvrez votre fichier `app/Controller/AppController.php` et ajoutez les lignes suivantes :

```
// app/Controller/AppController.php
class AppController extends Controller {
    //...

    public $components = array(
        'Session',
        'Auth' => array(
            'loginRedirect' => array('controller' => 'posts', 'action' => 'index'),
            'logoutRedirect' => array('controller' => 'pages', 'action' => 'display', 'home
        )
    );

    public function beforeFilter() {
        $this->Auth->allow('index', 'view');
    }
    //...
}
```

Il n'y a pas grand chose à configurer, puisque nous avons utilisé les conventions pour la table des users. Nous avons juste configurer les urls qui seront chargées après que la connexion et la déconnexion des actions sont effectuées, dans notre cas, respectivement à `/posts/` et `/`.

Ce que nous avons fait dans la fonction `beforeFilter` a été de dire au AuthComponent de ne pas exiger un login pour toutes les actions `index` et `view`, dans chaque controller. Nous voulons que nos visiteurs soient capables de lire et lister les entrées sans s'inscrire dans le site.

Maintenant, nous avons besoin d'être capable d'inscrire des nouveaux users, de sauvegarder leur nom d'user et mot de passe, et plus important de hasher leur mot de passe afin qu'il ne soit pas stocké en texte plain dans notre base de données. Disons à AuthComponent de laisser des users non-authentifiés d'accéder à la fonction add des users et de réaliser l'action connexion et deconnexion :

```
// app/Controller/UsersController.php

public function beforeFilter() {
    parent::beforeFilter();
    $this->Auth->allow('add'); // Laissons les users d'enregistrer eux-memes
}

public function login() {
    if ($this->request->is('post')) {
        if ($this->Auth->login()) {
            $this->redirect($this->Auth->redirect());
        } else {
            $this->Session->setFlash(__('Nom d\'user ou mot de passe invalide, réessayer'))
        }
    }
}

public function logout() {
    $this->redirect($this->Auth->logout());
}
```

Le hash du mot de passe n'est pas encore fait, ouvrez votre fichier de model `app/Model/User.php` et ajoutez ce qui suit :

```
// app/Model/User.php
App::uses('AuthComponent', 'Controller/Component');
class User extends AppModel {

// ...

public function beforeSave($options = array()) {
    if (isset($this->data[$this->alias]['password'])) {
        $this->data[$this->alias]['password'] = AuthComponent::password($this->data[$this->
    }
    return true;
}

// ...
```

Ainsi, maintenant à chaque fois qu'un user est sauvegardé, le mot de passe est hashé en utilisant le hashing fourni par défaut par la classe AuthComponent. Il nous manque juste un fichier template de vue pour la fonction de connexion, et le voilà :

```
<div class="users form">
<?php echo $this->Session->flash('auth'); ?>
<?php echo $this->Form->create('User');?>
    <fieldset>
        <legend><?php echo __('Merci de rentrer votre nom d\'user et mot de passe'); ?></le
```

```
        <?php echo $this->Form->input('username');
        echo $this->Form->input('password');
    ?>
    </fieldset>
<?php echo $this->Form->end(__('Connexion'));?>
</div>
```

Vous pouvez maintenant inscrire un nouvel user en rentrant l'url `/users/add` et vous connecter avec ce profil nouvellement créé en allant sur l'url `/users/login`. Essayez aussi d'aller sur n'importe quel url qui n'a pas été explicitement autorisée telle que `/posts/add`, vous verrez que l'application vous redirige automatiquement vers la page de connexion.

Et c'est tout ! Cela semble trop simple pour être vrai. Retournons en arrière un peu pour expliquer ce qui s'est passé. La fonction `beforeFilter` dit au component AuthComponent de ne pas exiger de connexion pour l'action `add` en plus des actions `index` and `view` qui étaient déjà autorisées dans la fonction `beforeFilter` de l'AppController.

L'action `login` appelle la fonction `$this->Auth->login()` dans AuthComponent, et cela fonctionne sans autre config car nous suivons les conventions comme mentionnées plus tôt. C'est-à-dire, avoir un model User avec les colonnes username et password, et utiliser un formulaire posté à un controller avec les données d'user. Cette fonction retourne si la connexion a réussi ou non, et en cas de succès, alors nous redirigeons l'user vers l'url configuré de redirection que nous utilisions quand nous avons ajouté AuthComponent à notre application.

La déconnexion fonctionne juste en allant à l'url `/users/logout` et redirigera l'user vers l'Url de Déconnexion configurée décrite précedemment. Cette url est le résultat de la fonction `AuthComponent::logout()` en cas de succès.

Autorisation (Qui est autorisé à accéder à quoi)

Comme mentionné avant, nous convertissons ce blog en un outil multi-user à autorisation, et pour ce faire, nous avons besoin de modifier un peu la table posts pour ajouter la référence au model User :

```
ALTER TABLE posts ADD COLUMN user_id INT(11);
```

Aussi, un petit changement dans PostsController est nécessaire pour stocker l'user connecté courant en référence pour le post créé :

```
// app/Controller/PostsController.php
public function add() {
    if ($this->request->is('post')) {
        $this->request->data['Post']['user_id'] = $this->Auth->user('id'); //Ligne ajoutée
        if ($this->Post->save($this->request->data)) {
            $this->Session->setFlash(__('Votre post a été sauvegardé.'));
            $this->redirect(array('action' => 'index'));
        }
    }
}
```

La fonction `user()` fournie par le component retourne toute colonne à partir de l'user connecté courant. Nous avons utilisé cette méthode pour ajouter les données dans les infos requêtées qui sont sauvegardées.

Sécurisons maintenant notre app pour empêcher certains auteurs de modifier ou supprimer les posts des autres. Des règles basiques pour notre app sont que les users admin peuvent accéder à tout url, alors que les users normaux (le role auteur) peuvent seulement accéder aux actions permises. Ouvrez encore la classe AppController et ajoutez un peu plus d'options à la config de Auth :

```
// app/Controller/AppController.php

public $components = array(
    'Session',
    'Auth' => array(
        'loginRedirect' => array('controller' => 'posts', 'action' => 'index'),
        'logoutRedirect' => array('controller' => 'pages', 'action' => 'display', 'home'),
        'authorize' => array('Controller') // Ligne ajoutée
    )
);

public function isAuthorized($user) {
    // Admin peut accéder à toute action
    if (isset($user['role']) && $user['role'] === 'admin') {
        return true;
    }

    // Refus par défaut
    return false;
}
```

Nous venons de créer un mécanisme très simple d'autorisation. Dans ce cas, les users avec le role `admin` sera capable d'accéder à tout url dans le site quand ils sont connectés, mais les autres (par ex le role `auteur`) ne peut rien faire d'autre par rapport aux users non connectés.

Ce n'est pas exactement ce que nous souhaitions, donc nous avons besoin de déterminer et fournir plus de règles à notre méthode `isAuthorized()`. Mais plutôt que de le faire dans AppController, déleguons à chaque controller la fourniture de ces règles supplémentaires. Les règles que nous allons ajouter à PostsController permettront aux auteurs de créer des posts mais empêcheront l'édition des posts si l'auteur ne correspond pas. Ouvrez le fichier `PostsController.php` et ajoutez le contenu suivant :

```
// app/Controller/PostsController.php

public function isAuthorized($user) {
    // Tous les users inscrits peuvent ajouter les posts
    if ($this->action === 'add') {
        return true;
    }

    // Le propriétaire du post peut l'éditer et le supprimer
    if (in_array($this->action, array('edit', 'delete'))) {
        $postId = $this->request->params['pass'][0];
        if ($this->Post->isOwnedBy($postId, $user['id'])) {
            return true;
        }
    }
```

```
    return parent::isAuthorized($user);
}
```

Nous surchargeons maintenant l'appel `isAuthorized()` de AppController's et vérifions à l'intérieur si la classe parente autorise déjà l'user. Si elle ne le fait pas, alors nous ajoutons juste l'autorisation d'accéder à l'action add, et éventuellement accés pour modifier et de supprimer. Une dernière chose à que nous avons oubliée d'exécuter est de dire si l'user à l'autorisation ou non de modifier le post, nous appelons une fonction `isOwnedBy()` dans le model Post. C'est généralement une bonne pratique de déplacer autant que possible la logique dans les models. Laissons la fonction s'exécuter :

```
// app/Model/Post.php

public function isOwnedBy($post, $user) {
    return $this->field('id', array('id' => $post, 'user_id' => $user)) === $post;
}
```

Ceci conclut notre tutoriel simple sur l'authentification et les autorisations. Pour sécuriser l'UsersController, vous pouvez suivre la même technique que nous faisions pour PostsController, vous pouvez aussi être plus créatif et coder quelque chose de plus général dans AppController basé sur vos propres règles.

Si vous avez besoin de plus de contrôle, nous vous suggérons de lire le guide complet Auth dans la section *Authentification* où vous en trouverez plus sur la configuration du component, la création de classes d'autorisation personnalisée, et bien plus encore.

Lectures suivantes suggérées

1. *Génération de code avec Bake* Génération basique CRUD de code
2. *Authentification* : Inscription d'user et connexion

Application Simple contrôlée par Acl

Note : Ce n'est pas un tutoriel pour débutants. Si vous commencez juste avec CakePHP, nous vous conseillons d'avoir un meilleur expérience d'ensemble des fonctionnalités du framework avant d'essayer ce tutoriel.

Dans ce tutoriel, vous allez créer une application simple avec *Authentification* et *Liste de contrôle d'accès (ACL)*. Ce tutoriel suppose que vous avez lu le tutoriel du *Tutoriel d'un Blog*, et que vous êtes familier avec *Génération de code avec Bake*. Vous devrez avoir un peu d'expérience avec CakePHP, et être familier avec les concepts MVC. Ce tutoriel est une briève introduction à `AuthComponent` et `AclComponent`.

Ce dont vous aurez besoin

1. Un serveur web opérationnel. Nous allons supposer que vous utilisez Apache, cependant les instructions pour utiliser d'autres serveurs devraient être très similaires. Nous pourrions avoir à jouer un peu avec la configuration du serveur, mais la plupart de gens peuvent se procurer CakePHP et le faire fonctionner sans qu'aucune configuration soit nécessaire.

2. Un serveur de base de données. Nous utiliserons MySQL dans ce tutoriel. Vous aurez besoin de connaître suffisamment de chose en SQL, notamment pour pouvoir créer une base de données : Cake prendra les rènes à partir d'ici.
3. Une connaissance des bases PHP. Plus vous aurez pratiqué la programmation orientée objet, mieux ça vaudra : mais n'ayez pas peur si vous êtes un fan de programmation procédurale.

Préparer notre Application

Premièrement, récupérons une copie récente de CakePHP

Pour obtenir un téléchargement à jour, visitez le projet CakePHP sur Github : https ://github.com/cakephp/cakephp/tags et téléchargez la version stable. Pour ce tutoriel vous aurez besoin de la dernière version 2.x.

Vous pouvez aussi dupliquer le dépôt en utilisant git (http ://git-scm.com/). `git clone git://github.com/cakephp/cakephp.git`.

Une fois que vous avez votre copie toute récente de Cake, configurez votre fichier "app/Config/database.php" et changez la valeur du Security.salt ("grain" de sécurité) dans votre fichier "app/Config/core.php". A ce stade, nous construirons un schéma simple de base de données sur lequel bâtir notre application. Exécutez les commandes SQL suivantes sur votre base de données :

```
CREATE TABLE users (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    password CHAR(40) NOT NULL,
    group_id INT(11) NOT NULL,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE groups (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE posts (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    user_id INT(11) NOT NULL,
    title VARCHAR(255) NOT NULL,
    body TEXT,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE widgets (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    part_no VARCHAR(12),
```

```
    quantity INT(11)
);
```

Ce sont les tables que nous utiliserons pour construire le reste de notre application. Une fois que nous avons la structure des tables dans notre base de données, nous pouvons commencer à cuisiner. Utilisez *Génération de code avec Bake* pour créer rapidement vos models, controllers et vues.

Pour utiliser cake bake, appelez "cake bake all" et cela listera les 4 tables que vous avez inseré dans mySQL. Séléctionnez "1. Group", et suivez ce qui est écrit sur l'écran. Répétez pour les 3 autres tables, et cela générera les 4 controllers, models et vues pour vous.

Evitez d'utiliser le Scaffold ici. La génération des ACOs en sera sérieusement affectée si vous cuisinez les controllers avec la fonctionnalité Scaffold.

Pendant la cuisson des Models, cake détectera auto-magiquement les associations entre vos Models (ou relations entre vos tables). Laissez Cake remplir les bonnes associations hasMany et belongsTo. Si vous êtes invité à choisir hasOne ou hasMany, d'une manière générale, vous aurez besoin d'une relation hasMany (seulement) pour ce tutoriel.

Laissez de côté les routing admin pour le moment, c'est déjà un assez compliqué sujet comme cela sans eux. Assurez-vous aussi de **ne pas** ajouter les Components Acl et Auth à aucun de vos controllers quand vous les cuisinez. Nous le ferons bien assez tôt. Vous devriez maintenant avoir des models, controllers, et des vues cuisinés pour vos users, groups, posts et widgets.

Préparer l'ajout d'Auth

Nous avons maintenant une application CRUD (Créer Lire Editer Supprimer) fonctionnelle. Bake devrait avoir mis en place toutes les relations dont nous avons besoin, si ce n'est pas le cas, faites-le maintenant. Il y a quelques autres éléments qui doivent être ajoutés avant de pouvoir ajouter les components Auth et Acl. Tout d'abord, ajoutez une action login et une action logout à votre `UsersController` :

```
public function login() {
    if ($this->request->is('post')) {
        if ($this->Auth->login()) {
            $this->redirect($this->Auth->redirect());
        } else {
            $this->Session->setFlash(__('Votre nom d\'user ou mot de passe sont incorrects
        }
    }
}

public function logout() {
    //Laissez vide pour le moment.
}
```

Ensuite créer le fichier de vue suivant pour la connexion `app/View/Users/login.ctp` :

```
echo $this->Form->create('User', array('action' => 'login'));
echo $this->Form->inputs(array(
    'legend' => __('Login'),
    'username',
```

```
    'password'
));
echo $this->Form->end('Connexion');
```

Ensuite nous devrons mettre à jour notre model User pour hasher les passwords avant qu'ils aillent dans la base de données. Stocker les passwords en brut est extrémement non sécurisé et AuthComponent va s'attendre à ce que vos passwords soient hashés. Dans `app/Model/User.php` ajoutez ce qui suit :

```
App::uses('AuthComponent', 'Controller/Component');
class User extends AppModel {
    // autre code.

    public function beforeSave($options = array()) {
        $this->data['User']['password'] = AuthComponent::password($this->data['User']['pass
        return true;
    }
}
```

Ensuite nous devons faire quelques modifications dans `AppController`. Si vous n'avez pas `/app/Controller/AppController.php`, créez le. Notez que cela va dans /app/Controller/, pas dans /app/app_controllers.php. Puisque nous voulons que notre site entier soit contrôllé avec Auth et Acl, nous allons les définir en haut dans `AppController` :

```
class AppController extends Controller {
    public $components = array(
        'Acl',
        'Auth' => array(
            'authorize' => array(
                'Actions' => array('actionPath' => 'controllers')
            )
        ),
        'Session'
    );
    public $helpers = array('Html', 'Form', 'Session');

    public function beforeFilter() {
        //Configure AuthComponent
        $this->Auth->loginAction = array('controller' => 'users', 'action' => 'login');
        $this->Auth->logoutRedirect = array('controller' => 'users', 'action' => 'login');
        $this->Auth->loginRedirect = array('controller' => 'posts', 'action' => 'add');
    }
}
```

Avant de configurer ACL, nous aurons besoin d'ajouter quelques users et groups. Avec `AuthComponent` en utilisation, nous ne serons pas capable d'accéder à aucune de nos actions, puisque nous ne sommes pas connectés. Nous allons maintenant ajouter quelques exceptions ainsi `AuthComponent` va nous autoriser à créer quelques groups et users. Dans les **deux**, votre `GroupsController` et votre `UsersController`, ajoutez ce qui suit :

```
public function beforeFilter() {
    parent::beforeFilter();
```

```
    // For CakePHP 2.0
    $this->Auth->allow('*');

    // For CakePHP 2.1 and up
    $this->Auth->allow();
}
```

Ces lignes disent à AuthComponent d'autoriser les accès publiques à toutes les actions. C'est seulement temporaire et ce sera retiré une fois que nous aurons quelques users et groups dans notre base de données. N'ajoutez pourtant encore aucun user ou group.

Initialiser les tables Acl dans la BdD

Avant de créer des users et groups, nous voulons les connecter à l'Acl. Cependant, nous n'avons pour le moment aucune table d'Acl et si vous essayez de visualiser les pages maintenant, vous aurez une erreur de table manquante ("Error : Database table acos for model Aco was not found."). Pour supprimer ces erreurs, nous devons exécuter un fichier de schéma. Dans un shell, exécutez la commande suivante :

```
./Console/cake schema create DbAcl
```

Ce schéma vous invite à supprimer et créer les tables. Répondez Oui (Yes) à la suppression et création des tables.

Si vous n'avez pas d'accès au shell, ou si vous avez des problèmes pour utiliser la console, vous pouvez exécuter le fichier sql se trouvant à l'emplacement suivant : /chemin/vers/votre/app/Config/Schema/db_acl.sql.

Avec les controllers configurés pour l'entrée de données et les tables Acl initialisées, nous sommes prêts à commencer, n'est-ce-pas ? Pas tout à fait, nous avons encore un peu de travail à faire dans les models users et groups. Concrètement, faire qu'ils s'attachent auto-magiquement à l'Acl.

Agir comme un requêteur

Pour que Auth et Acl fonctionnent correctement, nous devons associer nos users et groups dans les entrées de nos tables Acl. Pour ce faire, nous allons utiliser le behavior `AclBehavior`. Le behavior `AclBehavior` permet de connecter automagiquement des models avec les tables Acl. Son utilisation requiert l'implémentation de `parentNode()` dans vos models. Dans notre Model `User` nous allons ajouter le code suivant

```
class User extends Model {
    public $belongsTo = array('Group');
    public $actsAs = array('Acl' => array('type' => 'requester'));

    public function parentNode() {
        if (!$this->id && empty($this->data)) {
            return null;
        }
        if (isset($this->data['User']['group_id'])) {
            $groupId = $this->data['User']['group_id'];
        } else {
            $groupId = $this->field('group_id');
        }
```

```
        if (!$groupId) {
            return null;
        } else {
            return array('Group' => array('id' => $groupId));
        }
    }
}
```

Ensuite dans notre Model `Group` ajoutons ce qui suit :

```
class Group extends Model {
    public $actsAs = array('Acl' => array('type' => 'requester'));

    public function parentNode() {
        return null;
    }
}
```

Cela permet de lier les models `Group` et `User` à l'Acl, et de dire à CakePHP que chaque fois que l'on créé un User ou un Group, nous voulons également ajouter une entrée dans la table `aros`. Cela fait de la gestion des Acl un jeu d'enfant, puisque vos AROs se lient de façon transparente à vos tables `users` et `groups`. Ainsi, chaque fois que vous créez ou supprimez un groupe/user, la table Aro est mise à jour.

Nos controllers et models sont maintenant prêts à recevoir des données initiales et nos models `Group` et `User` sont reliés à la table Acl. Ajoutez donc quelques groups et users en utilisant les formulaires créés avec Bake en allant sur http ://exemple.com/groups/add et http ://exemple.com/users/add. J'ai créé les groups suivants :
– administrateurs
– managers
– users
J'ai également créé un user dans chaque groupe, de façon à avoir un user de chaque niveau d'accès pour les tests ultérieurs. Ecrivez tout sur du papier ou utilisez des mots de passe faciles, de façon à ne pas les oublier. Si vous faites un *SELECT * FROM aros ;'* depuis une commande mysql, vous devriez recevoir quelque chose comme cela :

```
+----+-----------+-------+-------------+-------+------+------+
| id | parent_id | model | foreign_key | alias | lft  | rght |
+----+-----------+-------+-------------+-------+------+------+
|  1 |      NULL | Group |           1 | NULL  |    1 |    4 |
|  2 |      NULL | Group |           2 | NULL  |    5 |    8 |
|  3 |      NULL | Group |           3 | NULL  |    9 |   12 |
|  4 |         1 | User  |           1 | NULL  |    2 |    3 |
|  5 |         2 | User  |           2 | NULL  |    6 |    7 |
|  6 |         3 | User  |           3 | NULL  |   10 |   11 |
+----+-----------+-------+-------------+-------+------+------+
6 rows in set (0.00 sec)
```

Cela nous montre que nous avons 3 groups et 3 users. Les users sont imbriqués dans les groups, ce qui signifie que nous pouvons définir des permissions sur une base par groupe ou par user.

ACL basé sur les groupe uniquement

Dans la cas où nous souhaiterions simplifier en utilisant les permissions par groups, nous avons besoin d'implémenter `bindNode()` dans le model `User` :

```
public function bindNode($user) {
    return array('model' => 'Group', 'foreign_key' => $user['User']['group_id']);
}
```

Cette méthode va demander à ACL de ne pas vérifier les AROs de `User` mais de seulement vérifier les AROs de `Group`.

Chaque user devra être assigné à un `group_id` pour que ceci fontionne correctement.

Dans ce cas, notre table *aros*' va ressembler à ceci

```
+----+-----------+-------+-------------+-------+------+------+
| id | parent_id | model | foreign_key | alias | lft  | rght |
+----+-----------+-------+-------------+-------+------+------+
|  1 |      NULL | Group |           1 | NULL  |    1 |    2 |
|  2 |      NULL | Group |           2 | NULL  |    3 |    4 |
|  3 |      NULL | Group |           3 | NULL  |    5 |    6 |
+----+-----------+-------+-------------+-------+------+------+
3 rows in set (0.00 sec)
```

Créer les ACOs (Access Control Objects)

Maintenant que nous avons nos users et groups (aros), nous pouvons commencer à intégrer nos controllers existants dans l'Acl et définir des permissions pour nos groups et users, et permettre la connexion / déconnexion.

Nos AROs sont automatiquement créés lorsque de nouveaux users et groups sont ajoutés. Qu'en est-t'il de l'auto-génération des ACOs pour nos controllers et leurs actions ? Et bien, il n'y a malheureusement pas de solution magique dans le core de CakePHP pour réaliser cela. Les classes du core offrent cependant quelques moyens pour créer manuellement les ACOs. Vous pouvez créer des objets ACO depuis le shell Acl, ou alors vous pouvez utiliser l'`AclComponent`. Créer les Acos depuis le shell ressemble à cela :

```
./Console/cake acl create aco root controllers
```

En utilisant l'AclComponent, cela ressemblera à :

```
$this->Acl->Aco->create(array('parent_id' => null, 'alias' => 'controllers'));
$this->Acl->Aco->save();
```

Ces deux exemples vont créer notre root ou ACO de plus haut niveau, qui sera appelé 'controllers'. L'objectif de ce nœud root est d'autoriser/interdire l'accès à l'échelle globale de l'application, et permet l'utilisation de l'Acl dans des objectifs non liés aux controllers/actions, tels que la vérification des permissions d'un enregistrement d'un model. Puisque nous allons utiliser un ACO root global, nous devons faire une petite modification à la configuration de `AuthComponent`. L'`AuthComponent` doit être renseigné sur l'existence de ce nœud root, de sorte que lors des contrôles de l'ACL, le component puisse utiliser le bon chemin

de nœud lors de la recherche controllers/actions. Dans l'`AppController`, assurez vous que le tableau `$components` contient l'`actionPath` défini avant.

```
class AppController extends Controller {
    public $components = array( 'Acl', 'Auth' => array(
            'authorize' => array( 'Actions' => array('actionPath' => 'controllers')
            )
        ), 'Session'
    );
```

Continuez à *Application Simple contrôlée par Acl - partie 2* pour continuer le tutoriel.

Application Simple contrôlée par Acl - partie 2

Un outil automatique pour la création des ACOs

Comme mentionné avant, il n'y a pas de façon pré-construite d'insérer tous vos controllers et actions dans Acl. Cependant, nous détestons tous faire des choses répétitives comme faire ce qui pourrait être des centaines d'actions dans une grande application.

Pour cela, il existe un plugin disponible très branché sur github, appelé AclExtras (https ://github.com/markstory/acl_extras/) qui peut être téléchargé sur La page de Téléchargements de Github (https ://github.com/markstory/acl_extras/zipball/master). Nous allons décrire brièvement la façon dont on l'utilise pour générer tous nos ACOs.

Premièrement prenez une copie du plugin et dézipper le ou dupliquer le en utilisant git dans *app/Plugin/AclExtras*. Ensuite, activez le plugin dans votre fichier *app/Config/boostrap.php* comme montré ci-dessus :

```
//app/Config/boostrap.php
// ...
CakePlugin::load('AclExtras');
```

Enfin exécutez la commande suivante dans la console de CakePHP :

```
./Console/cake AclExtras.AclExtras aco_sync
```

Vous pouvez récupérer un guide complet pour toutes les commandes disponibles comme ceci :

```
./Console/cake AclExtras.AclExtras -h
./Console/cake AclExtras.AclExtras aco_sync -h
```

Une fois remplie, votre table *acos* permet de créer les permissions de votre application.

Configurer les permissions

Pour créer les permissions, à l'image de la création des ACOs, il n'y a pas de solution magique et je n'en fournirai pas non plus. Pour autoriser des AROs à accéder à des ACOs depuis l'interface en ligne de commande, utilisez AclShell : Pour plus d'informations sur la façon de l'utiliser, consultez l'aide de AclShell que l'on peut avoir en lançant :

```
./Console/cake acl --help
```

Note : * a besoin d'être mis entre quotes ('*')

Pour donner des autorisations avec `AclComponent`, nous utiliserons la syntaxe de code suivante dans une méthode personnalisée :

```
$this->Acl->allow($aroAlias, $acoAlias);
```

Nous allons maintenant ajouter un peu d'autorisations/interdictions. Ajoutez ce qui suit à une fonction temporaire dans votre `UsersController` et visitez l'adresse dans votre navigateur pour les lancer (par ex : http ://localhost/cake/app/users/initdb). Si vous faîtes un `SELECT * FROM aros_acos`, vous devriez voir une pile entière de 1 et -1. Une fois que vous avez confirmé, vos permissions sont configurées, retirez la fonction :

```
public function beforeFilter() {
    parent::beforeFilter();
    $this->Auth->allow('initDB'); // Nous pouvons supprimer cette ligne après avoir fini
}

public function initDB() {
    $group = $this->User->Group;
    //Allow admins to everything
    $group->id = 1;
    $this->Acl->allow($group, 'controllers');

    //allow managers to posts and widgets
    $group->id = 2;
    $this->Acl->deny($group, 'controllers');
    $this->Acl->allow($group, 'controllers/Posts');
    $this->Acl->allow($group, 'controllers/Widgets');

    //allow users to only add and edit on posts and widgets
    $group->id = 3;
    $this->Acl->deny($group, 'controllers');
    $this->Acl->allow($group, 'controllers/Posts/add');
    $this->Acl->allow($group, 'controllers/Posts/edit');
    $this->Acl->allow($group, 'controllers/Widgets/add');
    $this->Acl->allow($group, 'controllers/Widgets/edit');
    //nous ajoutons un exit pour éviter d'avoir un message d'erreur affreux "missing views
    echo "tout est ok";
    exit;
}
```

Nous avons maintenant configurer quelques règles basiques. Nous avons autorisé les administrateurs pour tout. Les Manageurs peuvent accéder à tout dans posts et widgets. Alors que les users peuvent accéder aux add et edit des posts & widgets.

Nous devions avoir une référence d'un model de `Group` et modifier son id pour être capable de spécifier l'ARO que nous voulons, cela est dû à la façon dont fonctionne `AclBehavior`. `AclBehavior` ne configure pas le champ alias dans la table `aros` donc nous devons utiliser une référence d'objet ou un tableau pour référencer l'ARO que l'on souhaite.

Vous avez peut-être remarqué que j'ai délibérement oublié index et view des permissions Acl. Nous allons faire des actions publiques index et view dans `PostsController` et `WidgetsController`. Cela donne aux users non-autorisés l'autorisation de voir ces pages, en rendant ces pages publiques. Cependant, à tout moment, vous pouvez retirer ces actions des `AuthComponent::allowedActions` et les permissions pour view et edit seront les mêmes que celles dans Acl.

Maitenant, nous voulons enlever les références de `Auth->allowedActions` dans les controllers de vos users et groups. Ensuite ajouter ce qui suit aux controllers de vos posts et widgets :

```
public function beforeFilter() {
    parent::beforeFilter();
    $this->Auth->allow('index', 'view');
}
```

Cela enlève le "basculement à off" que nous avions mis plus tôt dans les controllers users et groups et cela rend public l'accès aux actions index et voir dans les controllers Posts et Widgets. Dans `AppController::beforeFilter()` ajoutez ce qui suit :

```
$this->Auth->allow('display');
```

Ce qui rend l'action 'display' publique. Cela rendra notre action PagesController : :display() publique. Ceci est important car le plus souvent le routage par défaut désigne cette action comme page d'accueil de votre application.

Connexion

Notre application est désormais sous contrôle d'accès, et toute tentative d'accès à des pages non publiques vous redirigera vers la page de connexion. Cependant, vous devrez créer une vue login avant que quelqu'un puisse se connecter. Ajoutez ce qui suit à `app/View/Users/login.ctp` si vous ne l'avez pas déjà fait :

```
<h2>Connexion</h2>
<?php
echo $this->Form->create('User', array('url' => array('controller' => 'users', 'action' =>
echo $this->Form->input('User.nom_user');
echo $this->Form->input('User.mot_de_passe');
echo $this->Form->end('Connexion');
?>
```

Si l'user est déjà connecté, on le redirige en ajoutant ceci au controller UsersController :

```
public function login() {
    if ($this->Session->read('Auth.User')) {
        $this->Session->setFlash('Vous êtes connecté!');
        $this->redirect('/');
    }
}
```

Vous devriez être maintenant capable de vous connecter et tout devrait fonctionner auto-maigiquement. Quand l'accès est refusé, les messages de Auth seront affichés si vous ajoutez le code `echo $this->Session->flash('auth')`.

Déconnexion

Abordons maintenant la déconnexion. Nous avions plus tôt laissé cette fonction vide, il est maintenant temps de la remplir. Dans `UsersController::logout()` ajoutez ce qui suit

```
$this->Session->setFlash('Au-revoir');
$this->redirect($this->Auth->logout());
```

Cela définit un message flash en Session et déconnecte l'User en utilisant la méthode logout de Auth. La méthode logout de Auth supprime tout simplement la clé d'authentification en session et retourne une url qui peut être utilisée dans une redirection. Si il y a d'autres données en sessions qui doivent être également effacées, ajoutez le code ici.

C'est fini !

Vous devriez maintenant avoir une application contrôlée par Auth et Acl. Les permissions Users sont définies au niveau du groupe, mais on peut également les définir en même temps par user. Vous pouvez également définir les permissions sur une base globale ou par controller et par action. De plus, vous avez un bloc de code réutilisable pour étendre facilement vos tables ACO lorsque votre application grandit.

CHAPITRE 13

Annexes

Les annexes contiennent des informations sur les nouvelles fonctionnalités de la version 2.0 ainsi qu'un guide de migration de la version 1.3 vers 2.0.

2.4 Guide de Migration

2.4 Guide de Migration

CakePHP 2.4 est une mise à jour complète à partir de l'API de 2.3. Cette page souligne les changements et améliorations faits dans 2.4.

Console

– Les messages de log de type notice seront maintenant en couleur dans les

terminaux qui supportent les couleurs.

SchemaShell

– `cake schema generate` supporte maintenant le paramètre `--exclude`.
– La constante `CAKEPHP_SHELL` est maintenant dépréciée et sera retirée dans

CakePHP 3.0.

BakeShell

– `cake bake model` permet maintenant la commande pour baker les `$behaviors`. Si les champs *lft*, *rght* et *parent_id* se trouvent dans votre table, cela va ajouter le behavior Tree, par exemple. Vous pouvez aussi étendre le ModelTask pour permettre à vos propres behaviors d'être reconnus.

– `cake bake` pour les views, models, controllers, tests et fixtures supportent maintenant les paramètres `-f` ou `--force` pour forcer l'écrasement de fichiers.

FixtureTask

– `cake bake fixture` supporte maintenant un paramètre `--schema` qui permet le baking de toutes les fixtures de façon noninteractive "all" tout en utilisant l'import du schema.

Core

Object

– `Object::log()` a un nouveau paramètre `$scope`.

Components

AuthComponent

– AuthComponent supporte maintenant le mode proper stateless lors de l'utilisation des authentifieurs *Basic* ou *Digest*. Partir de session peut être empêché en configurant `AuthComponent::$sessionKey` à false. Aussi maintenant lors de l'utilisation uniquement de *Basic* ou *Digest*, vous n'êtes plus redirigé vers la page de login. Pour plus d'infos, vérifiez la page `AuthComponent`.
– La propriété `AuthComponent::$authError` peut être définie au boléen *false* pour supprimer l'affichage du message flash.

PasswordHasher

– Les objets d'Authentification utilisent maintenant les nouveaux objets password hasher pour la génération et la vérification des password hashés Regardez *Hachage des mots de passe* pour plus d'info.

Model

Models

– `Model::save()`, `Model::saveField()`, `Model::saveAll()`, `Model::saveAssociated()`, `Model::saveMany()` prennent une nouvelle option `counterCache`. Vous pouvez la définir à false pour éviter de mettre à jour les valeurs du counter cache pour une opération de sauvegarde particulière.
– `Model::clear()` a été ajoutée.

Datasource

– Mysql, Postgres, et SQLserver supportent maintenant un tableau 'settings' dans la définition de connection. Cette paire de clé => valeur émettra des commandes `SET` lorque la connection est créée.

View

JsonView

– Le support de JSONP a été ajouté à `JsonView`.

HtmlHelper

– The API for `HtmlHelper::css()` has been been simplified. You can now provide an array of options as the second argument. When you do, the `rel` attribute defaults to 'stylesheet'.
– Une nouvelle option `escapeTitle` ajoutée à `HtmlHelper::link()` pour contrôler l'échappement seulement du titre du lien et pas des attributs.

TextHelper

– :php :meth : :*TextHelper : :autoParagraph()* a été ajoutée. Elle permet de convertir automatiquement les paragraphes de test en HTML.

PaginatorHelper

– :php :meth : :*PaginatorHelper : :param()* a été ajoutée.

Network

CakeRequest

– `CakeRequest::param()` a été ajoutée.
– `CakeRequest::is()` a été modifiée pour supporte un tableau de types et va retourner true si la requête correspond à tout type.
– `CakeRequest::isAll()` a été ajoutée pour vérifier qu'une requête correspond à tous les types donnés.

CakeResponse

– `CakeResponse::location()` a été ajoutée pour récupérer ou définir

l'en-tête de localisation du redirect.

CakeEmail

- Les messages de log d'email ont maintenant l'option `email` par défaut. Si vous ne voyez pas de contenus d'email dans vos logs, assurez-vous d'ajouter l'option `email` à votre configuration de log.
- `CakeEmail::emailPattern()` a été ajoutée. Cette méthode peut être utilisée pour faciliter les règles de validation d'email. C'est utile quand vous gérez certains hôtes Japonais qui permettent aux adresses non conformes d'être utilisées.
- `CakeEmail::attachments()` vous permet de fournir les contenus de fichier directement en utilisant la clé `data`.

HttpSocket

- `HttpSocket::patch()` a été ajoutée.

I18n

L10n

- `ell` est maintenant la locale par défaut pour le Grec comme spécifié par ISO 639-3 et `gre` son alias. Les dossiers de locale ont été ajustés en conséquence (de */Locale/gre/* en */Locale/ell/*).
- `fas` est maintenant la locale par défaut pour le Farsi comme spécifié par ISO 639-3 et `per` son alias. Les dossiers de locale ont été ajustés en conséquence (de */Locale/per/* en */Locale/fas/*).
- `sme` est maintenant la locale par défaut pour le Sami comme spécifié par ISO 639-3 et `smi` son alias. Les dossiers de locale ont été ajustés en conséquence (de */Locale/smi/* en */Locale/sme/*).
- `mkd` remplace `'mk` comme locale par défaut pour le Macedonien comme spécifié par ISO 639-3. Les dossiers de locale ont aussi été ajustés.
- Le code de Catalog `in` a été supprimé et remplacé par `id` (Indonesian), `e` a été supprimé et remplacé par `el` (Greek), `n` a été supprimé et remplacé par `nl` (Dutch), `p` a été supprimé et remplacé par `pl` (Polish), `sz` a été supprimé et remplacé par `se` (Sami).
- Kazakh a été ajouté `kaz` comme locale et `kk` comme code de catalog.
- Kalaallisut a été ajouté avec `kal` comme locale et `kl` comme code de catalog.

Logging

- Les moteurs de Log n'ont plus besoin du suffixe `Log` dans leur configuration. Donc pour le moteur de FileLog ; il suffit maintenant de définir `'engine' => 'File'`. Cela unifie la façon dont les moteurs sont nommés dans la configuration (regardez les moteurs de Cache par exemple). Note : Si vous avez un moteur de Log de type `'DatabaseLogger` qui ne suit pas les conventions, utilisez un suffix `Log` pour votre nom de classe, vous devez ajuster votre nom de classe en `DatabaseLog`. Vous devez aussi éviter les noms de classe comme `SomeLogLog` ce qui inclut le suffixe deux fois à la fin.

FileLog

- Deux nouvelles options de config `size` et `rotate` ont été ajoutées pour le moteur *FileLog*.

SyslogLog

– Le nouveau moteur de log *SyslogLog* a été ajouté pour streamer les messages au syslog.

Utility

General

– `pr` ne sort plus le HTML lors du lancement en mode cli.

Validation

– `Validation::date()` supporte maintenant les formats `y` et `ym`.
– Le code de pays de `Validation::phone()` pour le Canada a été changé de `can` en `ca` pour unifier les codes de pays pour les méthodes de validation selon ISO 3166 (codes à deux lettre).

CakeNumber

– Les monnaies `AUD`, `CAD` et `JPY` ont été ajoutées.
– Les symboles pour `GBP` et `EUR` sont maintenant UTF-8. Si vous mettez à jour une application non-UTF-8, assurez-vous que vous mettez à jour l'attribut statique `$_currencies` avec les symboles d'entité HTML appropriés (`£` et `€`) avant d'utiliser ces monnaies.

CakeTime

– `CakeTime::isPast()` et `CakeTime::isFuture()` ont été ajoutées.
– `CakeTime::timeAgoInWords()` a deux nouvelles options pour personnaliser les chaînes de sortie : `relativeString` (par défaut à `%s ago`) et `absoluteString` (par défaut à `on %s`).

Xml

– La nouvelle option `pretty` a été ajoutée à `Xml::fromArray()` pour retourner un Xml joliment formaté.

Error

ErrorHandler

– La nouvelle option de configuration `skipLog` a été ajoutée, qui permet d'échapper certains types d'Exception du Log. `Configure::write('Exception.skipLog', array('NotFoundException', 'ForbiddenException'));` vont éviter ces exceptions et celles qui les étendent d'être dans les logs quand la config `'Exception.log'` est à `true`

Routing

Router

– **`Router::baseUrl()` a été ajoutée en même temps que la valeur** `App.fullBaseURL` de Configure. Cette méthode remplace `FULL_BASE_URL` qui est maintenant dépréciée.

2.3 Guide de Migration

2.3 Guide de Migration

CakePHP 2.3 est une mise à jour de l'API complètement compatible à partir de 2.2. Cette page souligne les changements et les améliorations faits dans 2.3.

Constantes

Une application peut maintenant facileemnt définir `CACHE` et `LOGS`, puisqu'ils sont maintenant définis de façon conditionnelle par CakePHP.

Mise en Cache

– FileEngine est toujours le moteur de cache par défaut. Dans le passé, un certain nombre de personnes a des difficultés configurant et déployant APC correctement des deux façons en cli et web. L'utilisation des fichiers devrait rendre la configuration de CakePHP plus simple pour les nouveaux développeurs.

Component

AuthComponent

– Un nouvel adaptateur d'authenticate a été ajouté pour le support de hash blowfish/bcrypt hashed des mots de passe. Vous pouvez maintenant utiliser `Blowfish` dans votre tableau `$authenticate` pour permettre aux mots de passe bcrypt d'être utilisés.

PaginatorComponent

– PaginatorComponent support maintenant l'option `findType`. Ceci peut être utilisé pour spécifier quelle méthode find vous voulez utiliser pour la pagination. C'est un peu plus facile de manager et de définir que l'index 0ième.

SecurityComponent

- SecurityComponent supporte maintenant l'option `unlockedActions`. Ceci peut être utilisé pour désactiver toutes les vérifications de sécurité pour toute action listée dans cette option.

RequestHandlerComponent

- `RequestHandlerComponent::viewClassMap()` a été ajouté, qui est utilisé pour mapper un type vers le nom de classe de la vue. Vous pouvez ajouter `$settings['viewClassMap']` pour configurer automatiquement la viewClass correcte basée sur le type extension/content.

CookieComponent

- `CookieComponent::check()` a été ajoutée. Cette méthode fonctionne de la même façon que `CakeSession::check()`.

Console

- Le shell `server` a été ajouté. Vous pouvez utiliser cela pour commencer le serveur web PHP5.4 pour votre application CakePHP.
- Construire un nouveau projet avec bake définit maintenant le préfixe de cache de l'application avec le nom de l'application.

I18n

L10n

- `nld` est maintenant la locale par défaut pour Dutch comme spécifié par ISO 639-3 et `dut` pour ses alias. Les dossiers locale ont été ajustés pour cela (from */Locale/dut/* to */Locale/nld/*).
- Albanian est maintenant `sqi`, le Basque est maintenant `eus`, le Chinese est maintentant `zho`, Tibetan est maintenant `bod`, Czech est maintenant `ces`, Farsi est maitenant `fas`, French est maitenant `fra`, Icelandic est maitenant `isl`, Macedonian est maitenant `mkd`, Malaysian est maitenant `msa`, Romanian est maitenant `ron`, Serbian est maitenant `srp` et le Slovak est maitenant `slk`. Les dossiers locale correspondant ont été aussi ajustés.

Core

Configure

- `Configure::check()` a été ajouté. Cette méthode fonctionne de la manière que `CakeSession::check()`.

Error

Exceptions

– CakeBaseException a été ajouté, auquel toutes les Exceptions du coeur étendent. La classe d'Exception de base introduit aussi la méthode `responseHeader()` qui peut être appelée sur les instances d'Exception créees pour ajouter les headers à la réponse, puisque les Exceptions ne réutilisent pas toute instance de réponse.

Model

– Le support pour le type biginteger a été ajouté pour toutes les sources de données du coeur, et les fixtures.
– Support pour les indices `FULLTEXT` a été ajouté pour le driver MySQL.

Model

– `Model::find('list')` définit maintenant `recursive` basé sur le containment depth max ou la valeur récursive. Quand la liste est utilisée avec ContainableBehavior.

Validation

– Les méthodes de manque pour les validations vont **toujours** maintenant attraper les erreurs au lieu de le faire seulement en mode développement.

Network

SmtpTransport

– Le support TLS/SSL a été ajouté pour les connections SMTP.

CakeRequest

– `CakeRequest::onlyAllow()` a été ajouté.
– `CakeRequest::query()` a été ajouté.

CakeResponse

– `CakeResponse::file()` a été ajouté.

CakeEmail

– L'option `contentDisposition` a été ajoutée à `CakeEmail::attachments()`. Cela vous permet de désactiver le header Content-Disposition ajouté aux fichiers joints.

Routing

Router

– Support pour `tel:`, `sms:` ont été ajoutés à `Router::url()`.

View

– MediaView est déprécié, et vous pouvez maintenant utiliser les nouvelles fonctionnalités dans `CakeResponse` pour atteindre les mêmes résultats.
– La Serialization dans les vues Json et Xml ont été déplacés vers `_serialize()`
– Les callbacks beforeRender et afterRender sont maintenant appelés dans les vues Json et Xml quand on utilise les templates de vue.
– `View::fetch()` a maintenant un agument `$default`. Cet argument peut être utilisé pour fournir une valeur par défaut si un block doit être vide.
– `View::prepend()` a été ajouté pour permettre de mettre du contenu avant le block existant.

Helpers

FormHelper

– `FormHelper::select()` accèpte maintenant une liste de valeurs dans l'attribut disabled. Combiné avec `'multiple' => 'checkbox'`, cela vous permet de fournir une liste de valeurs que vous voulez désactiver.
– `FormHelper::postLink()` accèpte maintenant une clé `method`. Cela vous permet de créer des formulaires en lien en utilisant d'autres méthodes HTTP que POST.

TextHelper

– `TextHelper::tail()` a été ajouté pour tronquer le texte en commençant par la fin.
– *ending* dans `TextHelper::truncate()` est déprécié en faveur de *ellipsis*

Testing

– Une fixture du coeur par défaut pour la table `cake_sessions` a été ajoutée. Vous pouvez l'utiliser en ajoutant `core.cake_sessions` à votre liste de fixture.

Utility

CakeNumber

– `CakeNumber::fromReadableSize()` a été ajouté.
– `CakeNumber::formatDelta()` a été ajouté.

Folder

– `Folder::copy()` et `Folder::move()` supportent maintenant la possiblité de fusionner les répertoires de cible et de source en plus de sauter le suivant/écrire par dessus.

String

– `String::tail()` a été ajouté pour tronquer le texte en commençant par la fin.
– *ending* dans `String::truncate()` est déprécié en faveur de *ellipsis*

Debugger

– `Debugger::exportVar()` sort maintenant des propriétés private et protected dans PHP >= 5.3.0.

Security

– Le support pour bcrypt (http ://codahale.com/how-to-safely-store-a-password/) a été ajouté. Regardez la documentation de `Security::hash()` pour plus d'informations sur la façon d'utiliser bcrypt.

Validation

– `Validation::fileSize()` a été ajoutée

2.2 Guide de Migration

2.2 Guide de Migration

CakePHP 2.2 est une mise à jour de l'API complètement compatible à partir de 2.0/2.1. Cette page souligne les changements et améliorations faits dans 2.2.

Etapes requises pour mettre à jour

Quand on met à jour vers CakePHP 2.2, il est important d'ajouter quelques nouvelles valeurs de configuration dans `app/Config/bootstrap.php`. Les ajoutez va assurer que le behavior soit cohérent avec 2.1.x :

```
// Active les filtres du Dispatcher pour les assets du plugin, et
// du CacheHelper.
Configure::write('Dispatcher.filters', array(
    'AssetDispatcher',
    'CacheDispatcher'
));

// Ajouter la configuration logging.
```

```
CakeLog::config('debug', array(
    'engine' => 'FileLog',
    'types' => array('notice', 'info', 'debug'),
    'file' => 'debug',
));
CakeLog::config('error', array(
    'engine' => 'FileLog',
    'types' => array('warning', 'error', 'critical', 'alert', 'emergency'),
    'file' => 'error',
));
```

Vous devrez aussi modifier `app/Config/core.php`. Changez la valeur de `LOG_ERROR` en `LOG_ERR` :

```
define('LOG_ERROR', LOG_ERR);
```

Quand on utilise `Model::validateAssociated()` ou `Model::saveAssociated()` et les échecs de validation du model principal, les erreurs de validation des models associés ne sont plus détruites. `Model::$validationErrors` va maintenant toujours montrer toutes les erreurs. Vous aurez peut-être besoin de mettre à jour vos cas de test pour prendre en compte ce changement.

Console

I18N extract shell

– Une option a été ajoutée pour écraser les fichiers POT existant par défaut :

```
./Console/cake i18n extract --overwrite
```

Models

– `Model::_findCount()` va maintenant appeler les méthodes find personnalisées avec `$state = 'before'` et `$queryData['operation'] = 'count'`. Dans certains cas, les finds personnalisés retournent toujours le bon compte pour la pagination, mais la clé `'operation'` permet plus de flexibilité pour construire les autres requêtes, ou de supprimer les joins qui sont requis pour le finder personnnalisé lui-même. Puisque la pagination de méthodes find personnalisées ne fonctionne presque jamais, il y a besoin ce workarounds pour cela dans le niveau de model, qui ne sont plus necéssaires.

Datasources

– Les sources de données Dbo supportent maintenant les transactions réelles imbriquées. Si vous avez besoin d'utiliser cette fonctionnalité dans votre application, activez la en utilisant `ConnectionManager::getDataSource('default')->useNestedTransactions = true;`

Testing

- Le webrunner inclut maintenant des liens pour re-lancer un test avec la sortie en debug.
- Les cas de test générés pour Controller sont maintenant des sous-classes de `ControllerTestCase`.

Error Handling

- Quand les exceptions se répétent, ou que les exceptions sont levées quand on rend les pages d'erreur, le nouveau layout `error` sera utilisé. Il est recommandé de ne pas utiliser de helpers supplémentaires dans ce layout puisque il est là pour les erreurs de niveau développeur seulement. Cela règle les problèmes des erreurs fatales dans le rendu des pages d'erreur dû à l'utilisation du helper dans le layout `default`.
- Il est important de copier `app/View/Layouts/error.ctp` dans votre répertoire. Ne pas le faire ainsi mettra en échec le rendu des pages erreurs.
- Vous pouvez maintenant configurer la gestion d'erreur en console spécifique. En configurant `Error.consoleHandler`, et `Exception.consoleHandler` vous pouvez définir le callback qui va gérer les errors/exceptions levées dans les applications en console.
- Le gestionnaire configuré dans `Error.handler` et `Error.consoleHandler` va recevoir des codes d'erreur fatal (par ex : `E_ERROR`, `E_PARSE`, `E_USER_ERROR`).

Exceptions

- `NotImplementedException` a été ajouté.

Core

Configure

- `Configure::dump()` a été ajoutée. Elle est utilisée pour rendre les données de configuration persistentes dans des stockages durables comme des fichiers. Les deux `PhpReader` et `IniReader` fonctionnent avec elle.
- Un nouveau paramètre de config 'Config.timezone' est disponible que vous pouvez définir comme une chaîne de timezone d'utilisateur. Par ex, vous pouvez faire `Configure::write('Config.timezone', 'Europe/Paris')`. Si une méthode de la classe `CakeTime` est appelée avec le paramètre `$timezone` à null et 'Config.timezone' est défini, alors la valeur de 'Config.timezone' sera utilisée. Cette fonctionnalité vous permet de définir le timezone d'utilisateur juste une fois au lieu de le passer chaque fois dans les appels de fonction.

Controller

AuthComponent

- Les options pour les adapters définies dans `AuthComponent::$authenticate` accèptent maintenant une option `contain`. Ceci est utilisé pour définir des options de contenance pour le cas où les enregistrements de l'utilisateur sont chargés.

CookieComponent

- Vous pouvez maintenant crypter les valeurs de cookie avec le rijndael cipher. Ceci nécessite l'installation de l'extension mcrypt (http ://php.net/mcrypt). Utiliser rijndael donne aux valeurs du cookie le cryptage réel, et est recommandé à la place de XOR cipher disponible dans les versions précédentes. Le XOR cipher est toujours le schéma par défaut de cipher pour maintenir la compatibilité avec les versions précédentes. Vous pouvez en lire plus dans la documentation `Security::rijndael()`.

Pagination

- Paginer les finders personnalisés va maintenant retourner des comptes corrects, vois les changements de Model pour plus d'informations.

Network

CakeEmail

- `CakeEmail::charset()` et `CakeEmail::headerCharset()` ont été ajoutés.
- Les encodages Japonnais légaux sont maintenant gérés correctement. `ISO-2202-JP` est utilisé lorsque l'encodage est `ISO-2202-JP-MS` qui fonctionne autour d'un nombre de questions dans les mail clients quand il s'agit des encodages CP932 et Shift_JIS.
- `CakeEmail::theme()` a été ajoutée.
- `CakeEmail::domain()` a été ajoutée. Vous pouvez utiliser cette méthode pour définir le nom de domaine utilisé lors de l'envoi de mail à partir d'un script CLI ou si vous voulez contrôler le nom d'hôte utilisé pour envoyer l'email.
- Vous pouvez maintenant définir `theme` et `helpers` dans votre classe EmailConfig.

CakeRequest

- CakeRequest va maintenant automatiquement décoder les corps de requête `application/x-www-form-urlencoded` sur les requêtes `PUT` et `DELETE`. Ces données seront disponibles dans `$this->data` exactement comme les données POST le sont.

Utility

Set

- La classe `Set` est maintenant dépréciée, et remplacée par la classe `Hash`. Set ne sera pas retiré avant 3.0.
- `Set::expand()` a été ajoutée.

Hash

La classe `Hash` a été ajoutée dans 2.2. Elle remplace Set en fournissant une API plus cohérente, fiable et performante pour faire plusieurs des tâches que fait Set. Regardez la page *Hash* pour plus de détails.

CakeTime

- Le paramètre `$userOffset` a été remplacé par le paramètre `$timezone` dans toutes les fonctions pertinentes. Donc au lieu de la sortie numérique, vous pouvez maintenant passer une chaîne timezone ou un objet DateTimeZone. Passer les sorties numériques pour le paramètre `$timezone` est toujours possible pour une compatibilité rétro-active.
- `CakeTime::timeAgoInWords()` a l'option `accuracy` ajoutée. Cette option vous permet de spécifier la précision que doivent avoir les times formatés.
- Nouvelles méthodes ajoutées :
  - `CakeTime::toServer()`
  - `CakeTime::timezone()`
  - `CakeTime::listTimezones()`
- Le paramètre `$dateString` dans toutes les méthodes accèptent maintenant un objet DateTime.

Helpers

FormHelper

- FormHelper gère maintenant mieux l'ajout des classes requises aux entrées. Il honore maintenant la clé `on`.
- `FormHelper::radio()` supporte maintenant `empty` qui fonctionne de la même façon que l'option empty de `select()`.
- Ajout de `FormHelper::inputDefaults()` pour définir les propriétés habituelles pour chacune de ses entrées générées par le Helper.

TimeHelper

- Depuis 2.1, TimeHelper utilise la classe CakeTime pour toutes ses méthodes pertinentes. Le paramètre `$userOffset` a été remplacé par le paramètre `$timezone`.
- `TimeHelper::timeAgoInWords()` a l'option `element` ajoutée. Cela vous permet de spécifier un élément HTML pour entourer le time formaté.

HtmlHelper

- `HtmlHelper::tableHeaders()` supporte maintenant la configuration des attributs par cellule de table.

Routing

Dispatcher

- Les écouteurs d'Event peuvent maintenant être attachés aux appels du dispatcher, ceux-ci vont avoir la capacité de changer l'information de requête ou la réponse avant qu'elle soit envoyée au client. Vérifiez la documentation complète pour ces nouvelles fonctionnalités dans *Filtres du Dispatcher*
- Avec l'ajout de *Filtres du Dispatcher* vous aurez besoin de mettre à jour `app/Config/bootstrap.php`. Regardez *Etapes requises pour mettre à jour*.

Router

- `Router::setExtensions()` a été ajoutée. Avec la nouvelle méthode, vous pouvez maintenant ajouter plus d'extensions à parser, par exemple dans un fichier de routes de plugin.

Cache

Redis Engine

Un nouveau moteur de cache a été ajouté en utilisant phpredis extension (https ://github.com/nicolasff/phpredis) il est configuré de la même manière que le moteur Memcache.

Cache groups

Il est maintenant possible de tagger ou de labeliser les clés de cache sous les groupes. Cela facilite pour supprimer en masse les entrées associées mise en cache avec le même label. Les groupes sont déclarés au moment de la configuration quand on crée le moteur de cache :

```
Cache::config(array(
    'engine' => 'Redis',
    ...
    'groups' => array('post', 'comment', 'user')
));
```

Vous pouvez avoir autant de groupes que vous le souhaitez, mais gardez à l'esprit qu'ils ne peuvent pas être modifiés dynamiquement.

La méthode de la classe `Cache::clearGroup()` a été ajoutée. Elle prende le nom du groupe et supprime toutes les entrées labelisées avec la même chaîne.

Log

Les changements dans `CakeLog` requièrent maintenant une configuration supplémentaire dans votre `app/Config/bootstrap.php`. Regardez *Etapes requises pour mettre à jour*, et *Journalisation (logging)*.

- La classe `CakeLog` accèpte maintenant les mêmes niveaux de log que défini dans RFC 5424 (http ://tools.ietf.org/html/rfc5424). Plusieurs méthodes pratiques ont été aussi ajoutées :
  - `CakeLog::emergency($message, $scope = array())`
  - `CakeLog::alert($message, $scope = array())`
  - `CakeLog::critical($message, $scope = array())`
  - `CakeLog::error($message, $scope = array())`
  - `CakeLog::warning($message, $scope = array())`
  - `CakeLog::notice($message, $scope = array())`
  - `CakeLog::info($message, $scope = array())`
  - `CakeLog::debug($message, $scope = array())`
- Un troisième argument `$scope` a été ajouté à `CakeLog::write`. Regardez *Scopes de journalisation*.
- Un nouveau moteur de log : `ConsoleLog` a été ajouté.

Validation de Model

- Un nouvel objet `ModelValidator` a été ajouté pour déléguer le travail de validation des données du model, il est normalement transparent pour l'application et complètement rétro-compatible. Il fournit aussi une API riche pour ajouter, modifier et retirer les règles de validation. Vérifiez les docs pour cet objet dans *Validation des Donnnées*.
- Les fonctions de validation dans vos models devront avoir la visibilité "public" afin d'être accessibles par `ModelValidator`.
- De nouvelles règles de validation ont été ajoutées :
  - `Validation::naturalNumber()`
  - `Validation::mimeType()`
  - `Validation::uploadError()`

2.1 Guide de Migration

2.1 Guide de Migration

CakePHP 2.1 est une mise à jour de l'API complètement compatible à partir de 2.0. Cette page souligne les changements et améliorations faits pour 2.1.

AppController, AppHelper, AppModel et AppShell

Ces classes sont désormais tenues de faire partie du répertoire app, puisqu'elles ont été retirées du coeur de CakePHP. Si vous n'avez toujours pas ces classes, vous pouvez utiliser ce qui suit pour la mise à jour :

```
// app/View/Helper/AppHelper.php
App::uses('Helper', 'View');
class AppHelper extends Helper {
}

// app/Model/AppModel.php
App::uses('Model', 'Model');
class AppModel extends Model {
```

```
}

// app/Controller/AppController.php
App::uses('Controller', 'Controller');
class AppController extends Controller {
}

// app/Console/Command/AppShell.php
App::uses('Shell', 'Console');
class AppShell extends Shell {
}
```

Si votre application a ces fichiers/classes, vous n'avez rien besoin de faire. De plus, si vous utilisiez le PagesController du coeur, vous aurez aussi besoin de le copier dans votre répertoire app/Controller.

Fichiers .htaccess

Les fichiers `.htaccess` par défaut ont changé, vous devrez vous rappeler de les mettre à jour ou de mettre à jour les schémas URL de re-writing de vos serveurs web pour correspondre aux changements faits dans `.htaccess`

Models

- Le callback `beforeDelete` sera vidé avant les callbacks beforeDelete des behaviors. Cela donne plus de cohérence avec le reste des évènements déclenchés dans la couche Model.
- `Model::find('threaded')` accepte maintenant `$options['parent']` si vous utilisez un autre champ, alors `parent_id`. Aussi, si le model a TreeBehavior attaché et configuré avec un autre champ parent, le find threaded l'utilisera par défaut.
- Les paramètres pour les requêtes utilisant les requêtes préparées vont maintenant faire partie de l'instruction SQL.
- Les tableaux de validation peuvent maintenant être plus précis quand un champ est obligatoire. La clé `required` accepte `create` et `update`. Ces valeurs rendront un champ obligatoire lors de la création ou la mise à jour.
- Model now has a `schemaName` property. If your application switches datasources by modifying `Model::$useDbConfig` you should also modify `schemaName` or use `Model::setDataSource()` method which handles this for you. Le Model a maintenant une propriété `schemaName`. Si votre application change de sources de données en modifiant `Model::$useDbConfig`, vous devriez aussi modifier `schemaName` ou utiliser la méthode `Model::setDataSource()` qui gère cela pour vous.

CakeSession

Modifié dans la version 2.1.1 : CakeSession ne fixe plus l'en-tête P3P, puisque cela relève de la responsabilité de votre application. Pour plus d'informations, voyez le ticket #2515 (http ://cakephp.lighthouseapp.com/projects/42648/tickets/2515-cakephp-20-session-p3p-header-doesnt-work-in-an-iframe) dans la lighthouse

Behaviors

TranslateBehavior

– `I18nModel` a été déplacé vers un fichier séparé.

Exceptions

L'exception par défaut de rendu inclut maintenant des traces de pile plus détaillées y compris des extraits de fichiers et les décharges d'arguments pour toutes les fonctions dans la pile.

Utilitaire

Debugger

– `Debugger::getType()` a été ajoutée. Elle peut être utilisée pour récupérer le type de variables.
– `Debugger::exportVar()` a été modifiée pour créer une sortie plus lisible et plus utile.

debug()

debug() utilise maintenant `Debugger` en interne. Cela la rend plus cohérente avec avec Debugger, et profite des améliorations faîtes ici.

Set

– `Set::nest()` a été ajoutée. Elle prend en argument un tableau plat et retourne un tableau imbriqué.

File

– `File::info()` inclut les informations de taille et de mimetype du fichier.
– `File::mime()` a été ajoutée.

Cache

– `CacheEngine` a été déplacée dans un fichier séparé.

Configuration

– `ConfigReaderInterface` a été déplacée dans un fichier séparé.

App

- `App::build()` a maintenant la possibilité d'enregistrer de nouveaux paquets' à l'aide de''App : :REGISTER''. Voir *Ajoutez de nouveaux packages vers une application* pour plus d'informations.
- Les classes qui ne peuvent pas être trouvées sur les chemins configurés vont être cherchées dans `APP` comme un chemin de secours. Cela facilite le chargement automatique des répertoires imbriqués dans `app/Vendor`.

Console

Shell de Test

Un nouveau TestShell a été ajouté. Il réduit le typage requis pour exécuter les tests unitaires, et offre un chemin de fichier en fonction d'interface utilisateur

```
# Execute les tests du model post
Console/cake test app/Model/Post.php
Console/cake test app/Controller/PostsController.php
```

Le vieux shell testsuite et sa syntaxe sont encore disponibles.

Général

- Les fichiers générés ne contiennent plus les timestamps avec la génération des datetime.

Routing

Router

- Les routes peuvent maintenant utiliser une syntaxe spéciale `/**` pour inclure tous les arguments de fin en un argument unique passé . Voir la section sur *Connecter les Routes* pour plus d'informations.
- `Router::resourceMap()` a été ajoutée.
- `Router::defaultRouteClass()` a été ajoutée. Cette méthode vous autorise à définir la classe route par défaut utilisée pour toutes les routes à venir qui sont connectés.

Réseau

CakeRequest

- Ajout de `is('requested')` et `isRequested()` pour la détection de requestAction.

CakeResponse

- Ajout `CakeResponse::cookie()` pour la définition des cookies.
- Ajout d'un nombre de méthodes pour *Réglage fin du Cache HTTP*

Controller

Controller

- `Controller::$uses` a été modifié, la valeur par défaut est maintenant `true` à la place de false. De plus, les différentes valeurs sont traitées de façon légèrement différente, mais se comportera comme cela dans la plupart des cas.
  - `true` va charger le model par défaut et fusionnser avec AppController.
  - Un tableau va charger ces models et fusionner avec AppController.
  - Un tableau vide ne va charger aucun model, sauf ceux déclarés dans la classe de base.
  - `false` ne va charger aucun model, et ne va pas non plus fusionner avec la classe de base.

Components (Composants)

AuthComponent

- `AuthComponent::allow()` n'accepte plus `allow('*')` en joker pour toutes les actions. Utilisez juste `allow()`. Cela unifie l'API entre allow() et deny().
- L'option `recursive` a été ajoutée à toutes les cartes d'authentification. Vous permet de contrôler plus facilement les associations stockées dans la session.

AclComponent

- `AclComponent` ne met plus en minuscules et n'infléchit plus le nom de classe utilisé pour `Acl.classname`. A la place, il utilise la valeur fournie comme telle.
- Les implémentations Backend Acl devraient maintenant être mis dans `Controller/Component/Acl`.
- Les implémentations Acl doivent être déplacées dans le dossier Component/Acl à partir de Component. Par exemple si votre classe Acl a été appelée `CustomAclComponent`, et était dans `Controller/Component/CustomAclComponent.php`. Il doit être déplacé dans `Controller/Component/Acl/CustomAcl.php`, et être nommé `CustomAcl`.
- `DbAcl` a été déplacée dans un fichier séparé.
- `IniAcl` a été déplacée dans un fichier séparé.
- `AclInterface` a été déplacée dans un fichier séparé.

Helpers

TextHelper

- `TextHelper::autoLink()`, `TextHelper::autoLinkUrls()`, `TextHelper::autoLinkEmails()` echappe les inputs HTMS par défaut. Vous pouvez contrôler l'option `escape`.

HtmlHelper

- `HtmlHelper::script()` avait une option ajoutée `block`.
- `HtmlHelper::scriptBlock()` avait une option ajoutée `block`.
- `HtmlHelper::css()` avait une option ajoutée `block`.
- `HtmlHelper::meta()` avait une option ajoutée `block`.
- Le paramètre `$startText` de `HtmlHelper::getCrumbs()` peut maintenant être un tableau. Cela donne plus de contrôle et de flexibilité sur le premier lien crumb.
- `HtmlHelper::docType()` est par défaut html5.
- `HtmlHelper::image()` a maintenant une option `fullBase`.
- `HtmlHelper::media()` a été ajoutée. Vous pouvez utilisez cette méthode pour créer des éléments audio/vidéo HTML5.
- Le support du *syntaxe de plugin* a été ajouté pour `HtmlHelper::script()`, `HtmlHelper::css()`, `HtmlHelper::image()`. Vous pouvez maintenant faciliter les liens vers les assets des plugins en utilisant `Plugin.asset`.
- `HtmlHelper::getCrumbList()` a eu le paramètre `$startText` ajouté.

Vue

- `View::$output` est déprécié.
- `$content_for_layout` est déprécié. Utilisez `$this->fetch('content');` à la place.
- `$scripts_for_layout` est déprécié. Utilisez ce qui suit à la place :

```
echo $this->fetch('meta');
echo $this->fetch('css');
echo $this->fetch('script');
```

  `$scripts_for_layout` est toujours disponible, mais l'API *view blocks* donne un remplacement plus extensible et flexible.
- La syntaxe `Plugin.view` est maintenant disponible partout. Vous pouvez utiliser cette syntaxe n'importe où, vous réferencez le nom de la vue, du layout ou de l'elément.
- L'option `$options['plugin']` pour `element()` est déprécié. Vous devez utiliser `Plugin.nom_element` à la place.

Vues de type contenu

Deux nouvelles classes de vues ont été ajoutées à CakePHP. Une nouvelle classe `JsonView` et `XmlView` vous permettent de facilement générer des vues XML et JSON. Vous en apprendrez plus sur ces classes dans

la section *Vues JSON et XML*.

Vues étendues

`View` a une nouvelle méthode vous permettant d'enrouler ou 'étendre' une vue/élement/layout avec un autre fichier. Voir la section sur *Views étendues* pour plus d'informations sur cette fonctionnalité.

Thèmes

La classe `ThemeView` est dépreciée en faveur de la classe `View`. En mettant simplement `$this->theme = 'MonTheme'` activera le support theme et toutes les classes de vue qui étendaient `ThemeView` devront étendre `View`.

Blocks de Vue

Les blocks de Vue sont une façon flexible de créer des slots ou blocks dans vos vues. Les blocks remplacent `$scripts_for_layout` avec une API robuste et flexible. Voir la section sur *Utiliser les blocs de vues* pour plus d'informations.

Helpers

Nouveaux callbacks

Deux nouveaux callbacks ont été ajoutés aux Helpers. `Helper::beforeRenderFile()` et `Helper::afterRenderFile()`. Ces deux nouveaux callbacks sont déclenchés avant/après que chaque fragment de vue soit rendu. Cela inclut les éléments, layouts et vues.

CacheHelper

– Les tags `<!--nocache-->` fonctionnent maintenant correctement à l'intérieur des éléments.

FormHelper

– FormHelper omet désormais des champs désactivés à partir des champs hash sécurisés. Cela permet le fonctionnement avec `SecurityComponent` et désactive les inputs plus facilement.
– L'option `between` quand elle est utilisée dans le cas d'inputs radio, se comporte maintenant différemment. La valeur `between` est maintenant placée entre le légende et les premiers éléments inputs.
– L'option `hiddenField` avec les inputs checkbox peuvent maintenant être mis à une valeur spécifique comme 'N' plutôt que seulement 0.
– L'attribut `for` pour les inputs date et time reflètent maintenant le premier input généré. Cela peut impliquer que l'attribut for peut changer les inputs datetime générés.
– L'attribut `type` pour `FormHelper::button()` peut maintenant être retiré. Il met toujours 'submit' par défaut.

– `FormHelper::radio()` vous permet maintenant de désactiver toutes les options. Vous pouvez le faire en mettant soit `'disabled' => true` soit `'disabled' => 'disabled'` dans le tableau `$attributes`.

PaginatorHelper

– `PaginatorHelper::numbers()` a maintenant une option `currentClass`.

Testing

– Les Web test runner affichent maintenant le numéro de version de PHPUnit.
– Les Web test runner configurent par défaut l'affichage des test des app.
– Les Fixtures peuvent être créées pour différentes sources de données autre que $test.
– Les Models chargés utilisant la ClassRegistry et utilisant une autre source de données aura son nom de source donnée préfixé par `test_` (ex : source de données *master* essaiera d'utiliser *test_master* dans la testsuite)
– Les cas de Test sont générés avec des méthodes de configuration de la classe spécifique.

Evénements

– Un nouveau système générique des évènements a été construit et a remplacé la façon dont les callbacks ont été dispatchés. Cela ne devrait représenter aucun changement dans votre code.
– Vous pouvez envoyer vos propres évènements et leur attacher des callbacks selon vos souhaits, utile pour la communication inter-plugin et facilite le découplage de vos classes.

Nouvelles caractéristiques dans CakePHP 2.1

Models

Model : :saveAll(), Model : :saveAssociated(), Model : :validateAssociated()

`Model::saveAll()` et ses amis supportent maintenant le passement de *fieldList* pour de multiples modèles. Exemple :

```
$this->SomeModel->saveAll($data, array(
    'fieldList' => array(
        'SomeModel' => array('field_1'),
        'AssociatedModel' => array('field_2', 'field_3')
    )
));
```

`Model::saveAll()` et ses amis peuvent maintenant sauvegarder sur des niveaux de profondeur illimités. Exemple :

```
$data = array(
    'Article' => array('title' => 'My first article'),
    'Comment' => array(
        array('body' => 'Comment 1', 'user_id' => 1),
        array('body' => 'Save a new user as well', 'User' => array('first' => 'mad', 'last'
    ),
);
$this->SomeModel->saveAll($data, array('deep' => true));
```

View

View Blocks (Blocks de Vue)

View Blocks sont un mécanisme permettant l'inclusion de slots de contenu, en autorisant les classes enfants de vue ou les éléments à fournir le contenu personnalisé pour ce block.

Les blocks sont sortis en appelant la méthode `fetch` sur `View`. Par exemple, ce qui suit peut être placé dans votre fichier `View/Layouts/default.ctp` :

```
<?php echo $this->fetch('my_block'); ?>
```

Cela affichera le contenu du block si disponible, ou ue chaîne de caractère vide si elle n'est pas définie.

Définir le contenu d'un block peut être fait de plusieurs façons. Une simple attribution de donnée peut être faite en utilisant *assign* :

```
<?php $this->assign('my_block', 'Hello Block'); ?>
```

Ou vous pouvez l'utiliser pour capturer une section de contenu plus complexe :

```
<?php $this->start('my_block'); ?>
    <h1>Hello Block!</h1>
    <p>Ceci est block de contenu</p>
    <p>Page title: <?php echo $title_for_layout; ?></p>
<?php $this->end(); ?>
```

Le Block capturant aussi le support d'imbrication :

```
<?php $this->start('my_block'); ?>
    <h1>Hello Block!</h1>
    <p>This is a block of content</p>
    <?php $this->start('second_block'); ?>
        <p>Page title: <?php echo $title_for_layout; ?></p>
    <?php $this->end(); ?>
<?php $this->end(); ?>
```

ThemeView

Dans 2.1, l'utilisation de `ThemeView` est dépréciée en faveur de l'utilisation de la classe `View` elle-même. `ThemeView` est maintenant une classe stub.

All custom pathing code has been moved into the `View` class, meaning that it is now possible for classes extending the `View` class to automatically support themes. Whereas before we might set the `$viewClass` Controller property to `Theme`, it is now possible to enable themes by simply setting the `$theme` property. Example :

```
App::uses('Controller', 'Controller');

class AppController extends Controller {
    public $theme = 'Exemple';
}
```

Toutes les classe View qui étendent `ThemeView` dans 2.0 doivent maintenant simplement étendre `View`.

JsonView

Une nouvelle classe qui facilite la sortie de contenu JSON.

Précédemment, il était nécessaire de créer un layout JSON (`APP/View/Layouts/json/default.ctp`) et une vue correspondante pour chaque action qui sortirait le JSON. Ceci n'est plus requis avec `JsonView`.

`JsonView` est utilisée comme tout autre classe de vue, en la définissant sur le controller. Exemple :

```
App::uses('Controller', 'Controller');

class AppController extends Controller {
    public $viewClass = 'Json';
}
```

Une fois que vous avez configuré le controller, vous avez besoin d'identifier quel contenu devrait être sérialisé en JSON, en paramètrant la variable vue `_serialize`. Exemple :

```
$this->set(compact('users', 'posts', 'tags'));
$this->set('_serialize', array('users', 'posts'));
```

L'exemple ci-dessus résulterait seulement dans les variables `users` et `posts`, étant sérialisé pour la sortie JSON, comme ceci :

```
{"users": [...], "posts": [...]}
```

Il n'y a plus aucun besoin de créer des fichiers de vue `ctp` afin d'afficher le contenu Json.

La personnalisation future de la sortie peut être atteinte en étendant la classe `JsonView` avec votre propre classe de vue personnalisée si requise.

Les exemples suivants entourent le résultat avec `{results: ... }` :

```
App::uses('JsonView', 'View');
class ResultsJsonView extends JsonView {
    public function render($view = null, $layout = null) {
        $result = parent::render($view, $layout);
        if (isset($this->viewVars['_serialize'])) {
            return json_encode(array('results' => json_decode($result)));
        }
        return $result;
    }
}
```

XmlView

Un peu comme `JsonView`, `XmlView` requièrt que vous configuriez la variable de vue `_serialize` afin d'indiquer quelle information serait sérialisée en XML pour la sortie.

$this->set(compact('users', 'posts', 'tags')) ; $this->set('_serialize', array('users', 'posts')) ;

L'exemple ci-dessus résulterait dans seulement les variables `users` et `posts` étant sérialisées pour la sortie XML, comme ceci :

```
<response><users>...</users><posts>...</posts></response>
```

Notez que XmlView ajoute un noeud de `response` pour entourer tout contenu sérialisé.

Rendu de Vue conditionnel

Plusieurs nouvelles méthodes ont été ajoutées à `CakeRequest` pour faciliter la tâche de paramètrer les headers HTTP corrects en mettant le HTTP en cache. Vous pouvez maintenant définir notre stratégie de mise en cache en utilisant l'expiration ou la validation HTTP du cache du model, ou de combiner les deux. Maintenant, il y a des méthodes spécifiques dans `CakeRequest` to fine-tune Cache-Control directives, set the entity tag (Etag), set the Last-Modified time and much more.

Quand ces méthodes sont combinés avec le `RequestHandlerComponent` activé dans votre controller, le component décidera automatiquement si la réponse est déjà mise en cache dans le client et enverra un code de statut *304 Not Modified* avant le rendu de la vue. Sauter le processus de rendu de vue sauvegarde les cycles CPU et la mémoire. :

```
class ArticlesController extends AppController {
    public $components = array('RequestHandler');

    public function view($id) {
        $article = $this->Article->read(null, $id);
        $this->response->modified($article['Article']['modified']);
        $this->set(compact('article'));
    }
}
```

Dans l'exemple ci-dessus, la vue ne sera pas rendu si le client envoie le header *If-Modified-Since*, et la réponse aura un statut 304.

Helpers

Pour faciliter l'utilisation en dehors de la couche `View`, les méthodes des helpers `TimeHelper`, `TextHelper`, et `NumberHelper` ont été extraites respectivement des classes `CakeTime`, `String`, et `CakeNumber`.

Pour utiliser les nouvelles classes utilitaires :

```
class AppController extends Controller {

    public function log($msg) {
        $msg .= String::truncate($msg, 100);
        parent::log($msg);
    }
}
```

Vous pouvez écraser la classe par défaut à utiliser en créant une nouvelle classe dans votre dossier `APP/Utility`, par exemple : `Utility/MyAwesomeStringClass.php`, et le spécifier dans la clé `engine` :

```
// Utility/MyAwesomeStringClass.php
class MyAwesomeStringClass extends String {
    // mon truchement est meilleur que les votres
    public function static truncate($text, $length = 100, $options = array()) {
        return null;
    }
}

// Controller/AppController.php
class AppController extends Controller {
    public $helpers = array(
        'Text' => array(
            'engine' => 'MyAwesomeStringClass',
            ),
        );
}
```

HtmlHelper

Une nouvelle fonction `HtmlHelper::media()` a été ajoutée pour la génération d'éléments HTML audio/video.

2.0 Guide de Migration

Guide de Migration 2.0

Cette page résume les changements par rapport à CakePHP 1.3 qui aidera pour les projets de migration vers la version 2.0, ainsi qu'une référence pour se mettre à jour sur les changements faits dans le coeur

depuis la branche CakePHP 1.3. Assurez vous de lire les autres pages de ce guide pour toutes les nouvelles fonctionnalités et les changements de l'API.

Astuce : Faites bien un checkout *Mise à jour shell* inclu dans le coeur de la 2.0 pour vous aider à migrer du code de la 1.3 à la 2.0.

Support des Versions de PHP

CakePHP 2.x supporte la Version de PHP 5.2.8 et supérieur. Le support de PHP4 a été supprimé. Pour les développeurs qui travaillent avec un environnement de production PHP4, les versions de CakePHP 1.x continuent le support de PHP4 pour la durée de vie de leur développement.

Le passage à PHP5 siginifie que toutes les méthodes et propriétés ont été mises à jour avec les mots-clés correspondants. Si votre code tente d'accéder à des méthodes privées ou protégées avec une étendue public, vous rencontrerez des erreurs.

Bien que cela ne constitue pas un changement énormer du framework, cela signifie qu'un accès aux méthodes et variables à la visibilité serrée n'est maintenant plus possible.

Le nommage des Fichiers et Dossiers

Dans CakePHP 2.0, nous avons repensé la façon de structurer nos fichiers et dossiers. Etant donné que PHP 5.3 supporte les espaces de nom (namespaces), nous avons décidé de préparer notre base de code pour l'adoption dans un futur proche de cette version de PHP, donc nous avons adopté https ://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md. Tout d'abord, nous avons regardé la structure interne de CakePHP 1.3 et avons réalisé qu'après toutes ces années, il n'y avait ni organisation claire des fichiers, ni une structure de dossiers vraiment logique où chaque fichier se trouve où il devrait. Avec ce changement, nous serions autorisés à expérimenter un peu le chargement (presque) automatique des classes pour augmenter les performances globales du framework.

Le plus grand obstacle pour réussir cela, était de maintenir une sorte de compatiblité rétro-active avec la façon dont les classes sont chargées en ce moment, et nous ne voulions définitivement pas devenir un framework avec des énormes préfixes de classe, des noms de classe du type `Mon_Enorme_Classe_Dans_Le_Progiciel`. Nous avons décidé d'adopter une stratégie de garder des noms de classe simples, tout en offrant une façon très intuitive de déclaration des emplacements de classe et des chemins de migration clairs pour la future version PHP 5.3 de CakePHP. Tout d'abord, mettons en évidence les principaux changements dans la standardisation du nommage des fichiers que nous avons adoptée :

Noms des Fichiers

Tous les fichiers contenant les classes doivent être nommés selon la classe qu'il contiennent. Aucun fichier ne doit contenir plus d'une classe. Donc, plus de minuscules ou de soulignements dans les noms de fichier. Voici quelques exemples :
– `mes_trucs_controller.php` devient `MesTrucsController.php`
– `form.php` (un Helper) devient `FormHelper.php`

– `session.php` (un Component) devient `SessionComponent.php`
Cela rend le nommage des fichiers beaucoup plus clair et cohérent à travers les applications, et aussi évite quelques cas où le chargement des fichiers aurait pu été géné dans le passé et aurait pu entrainé un chargement non souhaité de fichiers.

Les Noms des Dossiers

La plupart des dossiers devront être en CamelCase, spécialement ceux contenant des classes. En songeant aux espaces de noms, chaque dossier représente un niveau dans la hiérachie des espaces de noms, les dossiers qui ne contiennent pas de classes, ou ne constituent pas un espace de noms sur eux-mêmes, devraient être en LowerCase.

Dossiers en CamelCase :
– Config
– Console
– Controller
– Controller/Component
– Lib
– Locale
– Model
– Model/Behavior
– Plugin
– Test
– Vendor
– View
– View/Helper
Dossiers en LowerCase :
– tmp
– webroot

htaccess (URL Rewriting)

Dans votre fichier `app/webroot/.htaccess` remplacez le `RewriteRule ^(.*)$ index.php?url=$1 [QSA,L]` avec `RewriteRule ^(.*)$ index.php?/$1 [QSA,L]`

AppController / AppModel / AppHelper / AppShell

Les fichiers `app/app_controller.php`, `app/app_model.php`, `app/app_helper.php` sont situés et nommés respectivement comme ceci `app/Controller/AppController.php`, `app/Model/AppModel.php` et `app/View/Helper/AppHelper.php`.

Aussi, les shell/task sont étendus (extend) Appshell. Vous pouvez avoir votre propre AppShell.php dans `app/Console/Command/AppShell.php`.

Internationalization / Localization

`__()` (La fonction raccourci de Double underscore) retourne toujours la traduction (plus de echo).

Si vous voulez changer les résultats de la traduction, utilisez :

```
echo __('Mon Message');
```

Cela remplace toutes les méthodes de traduction raccourcies :

```
__()
__n()
__d()
__dn()
__dc()
__dcn()
__c()
```

A côté de cela, si vous passez des paramètres supplémentaires, la traduction appelera sprintf (http ://php.net/manual/en/function.sprintf.php) avec ces paramètres retournés précédemment avant de retourner. Par exemple :

```
// Retournera quelque chose comme "Appelé: MaClasse:maMethode"
echo __('Appelé: %s:%s', $nomdelaclasse, $nomdelamethode);
```

Elle est valide pour toutes les méthodes raccourcies de traduction.

Plus d'informations sur les spécificités de la fonction : sprintf (http ://php.net/manual/en/function.sprintf.php).

Emplacement de la Classe et constantes changées

Les constantes `APP` et `CORE_PATH` ont des valeur cohérentes entre le web et les environnement de la console. Dans les précedentes versions de CakePHP, ces valeurs changeaient selon l'environnement.

Basics.php

- `getMicrotime()` a été retirée. Utilisez la fonction native `microtime(true)` à la place.
- `e()` a été retirée. Utilisez `echo`.
- `r()` a été retirée. Utilisez `str_replace`.
- `a()` a été retirée. Utilisez `array()`
- `aa()` a été retirée. Utilisez `array()`
- `up()` a été retirée. Utilisez `strtoupper()`
- `low()` a été retirée. Utilisez `strtolower()`
- `params()` a été retirée. Il n'était utilisé nul part dans CakePHP
- `ife()` a été retirée. Utilisez un opérateur ternaire.
- `uses()` a été retirée. Utilisez `App::import()` à la place.
- La compatibilité des fonctions de PHP4 a été retirée.
- La constante PHP5 a été retirée.

– La variable Globale appelée `$TIME_START` a été retirée. Utilisez la constante `TIME_START` ou `$_SERVER['REQUEST_TIME']` à la place.

Constantes Retirées

Un nombre de constantes ont été retirées, puisqu'elles n'ataient plus exactes ou bien étaient dupliquées.
– APP_PATH
– BEHAVIORS
– COMPONENTS
– CONFIGS
– CONSOLE_LIBS
– CONTROLLERS
– CONTROLLER_TESTS
– ELEMENTS
– HELPERS
– HELPER_TESTS
– LAYOUTS
– LIB_TESTS
– LIBS
– MODELS
– MODEL_TESTS
– SCRIPTS
– VIEWS

CakeRequest

Cette nouvelle classe encapsule les paramètres et fonctions liées aux requêtes entrantes. Elle remplace plusieurs fonctionnalités de `Dispatcher`, `RequestHandlerComponent` et Controller. Elle remplace aussi le tableau `$this->params` à tout endroit. `CakeRequest` implémente `ArrayAccess` donc la plupart des interactions avec les anciens tableaux params n'ont pas besoin de changement. Voir les nouvelles fonctionnalités de CakeRequest pour plus d'informations.

Gestion des Requêtes, $_GET['url'] et fichiers .htaccess

CakePHP n'utilise plus `$_GET['url']` pour la gestion des chemins des requêtes de l'application. A la place il utilise `$_SERVER['PATH_INFO']`. Cela fournit une façon plus uniforme de gestion des requêtes entre les serveurs avec URL rewriting et ceux sans. Du fait de ces changements, vous aurez besoin de mettre à jour vos fichiers .htaccess et `app/webroot/index.php`, puisque ces fichiers ont été changés pour accueillir les changements. De plus, `$this->params['url']['url']` n'existe plus. A la place, vous devrez utiliser $this->request->url pour accéder à la même valeur. Cet attribut contient maintenant l'url sans slash `/` au début.

Note : Pour la page d'accueil elle-même (`http://domain/`) $this->request->url retourne maintenant le boléen `false` au lieu de `/`. Assurez-vous de vérifier cela de cette façon :

```
if (!$this->request->url) {} // au lieu de $this->request->url === '/'
```

Components (Composants)

Component est maintenant la classe de base requise pour tous les components (components). Vous devrez mettre à jour vos components et leurs constructeurs, puisque tous deux ont changé :

```
class PrgComponent extends Component {
    public function __construct(ComponentCollection $collection, $settings = array()) {
        parent::__construct($collection, $settings);
    }
}
```

Tout comme les helpers il est important d'appeler `parent::__construct()` dans les components avec les constructeurs surchargés. Les paramètres pour un component sont aussi maintenant passés à travers le constructeur, et non plus via le callback `initialize()`. Cela aide à avoir de bons objets construits, et autorise la classe de base à gérer les propriétés supérieures.

Depuis que les paramètres ont été déplacés au constructeur du component, le callback `initialize()` ne reçoit plus `$settings` en 2ème paramètre. Vous devrez mettre à jour vos components pour utiliser la signature méthode suivante :

```
public function initialize($controller) { }
```

De plus, la méthode initialize() est seulement appelée sur les components qui sont permis. Cela signifie en général que les components qui sont directement attachés à l'objet controller.

Callbacks dépréciés supprimés

Tous les callbacks dépréciés dans Component ont été transférés à ComponentCollection. A la place, vous devriez utiliser la méthode *trigger()* pour intéragir avec les callbacks. Si vous avez besoin de déclencher un callback, vous pouvez le faire en appelant :

```
$this->Components->trigger('someCallback', array(&$this));
```

Changement dans la désactivation des components

Dans le passé, vous étiez capable de désactiver les components via *$this->Auth->enabled = false ;* par exemple. Dans CakePHP 2.0 vous devriez utiliser la méthode de désactivation des ComponentCollection's, *$this->Components->disable('Auth') ;*. Utiliser les propriétés actives ne va pas fonctionner.

AclComponent

- Les implémentations `AclComponent` sont maintenant requises pour implémenter `AclInterface`.
- `AclComponent::adapter()` a été ajouté pour permettre l'éxecution de la modification de l'utilisation de l'implémentation du component `ACL`.

– `AclComponent::grant()` a été déprécié, il sera supprimé dans une version future. Utilisez `AclComponent::allow()` à la place.
– `AclComponent::revoke()` a été déprécié, il sera supprimé dans une version future. Utilisez AclComponent : :deny() à la place.

RequestHandlerComponent

Beaucoup de méthodes de RequestHandlerComponent sont justes des proxies pour les méthodes de `CakeRequest`. Le méthodes suivantes ont été dépréciées et seront retirées dans les versions futures :
– `isSsl()`
– `isAjax()`
– `isPost()`
– `isPut()`
– `isFlash()`
– `isDelete()`
– `getReferer()`
– `getClientIp()`
– **`accepts()`, `prefers()`, `requestedWith()` Tous sont maintenant gérés** dans les types de contenu. Ils ne fonctionnent plus avec les mime-types. Vous pouvez utiliser `RequestHandler::setContent()` pour créer des nouveaux types de contenu.
– **`RequestHandler::setContent()` n'accepte plus de tableau en tant** qu'argument unique, vous devez fournir les deux arguments.

SecurityComponent

SecurityComponent ne gère plus l'Authentification Basic et Sommaire (Digest). Elles sont toutes deux gérées par le nouveau AuthComponent. Les méthodes suivantes ont été retirées de SecurityComponent :
– requireLogin()
– generateDigestResponseHash()
– loginCredentials()
– loginRequest()
– parseDigestAuthData()
De plus les propriétés suivantes ont été retirées :
– $loginUsers
– $requireLogin
Le déplacement des fonctionalités verss Authcomponent a été faite pour fournir un endroit unique pour tous les types d'authentification et pour rationaliser les rôles de chaque component.

AuthComponent

AuthComponent a été entièrement refait dans 2.0, ça a été fait pour réduire les confusions et frustrations des développeurs. De plus, AuthComponent a été construit plus flexible et extensible. Vous pouvez trouver plus d'informations dans le guide *Authentification*.

EmailComponent

EmailComponent a été déprecié et a été crée une nouvelle classe de librairie pour envoyer les emails. Voir les changements pour Email *CakeEmail* pour plus de détails.

SessionComponent

SessionComponent a perdu les méthodes suivantes.
– activate()
– active()
– __start()

Retrait de cakeError

La méthode `cakeError()` a été retirée. Il est recommandé que vous changiez toutes les utilisations de `cakeError` pour utiliser les exceptions. `cakeError` a été retirée car elle simulait les exceptions. Plutôt que la simulation, de réelles exceptions sont utilisées dans CakePHP 2.0.

Gestion des Erreurs

L'implémentation de la gestion des erreurs a changé de façon spectaculaire dans 2.0. Les exceptions ont été introduites partout dans le framework, et la gestion des erreurs a été mise à jour pour offrir plus de contrôle et de flexibilité. Vous pouvez en lire plus dans les sections *Exceptions* et *Gestion des Erreurs*.

Classes Lib

App

L'API pour `App::build()` a changé pour `App::build($paths, $mode)`. Elle vous autorise maintenant à soit ajouter, soit faire précéder ou bien réinitialiser / remplacer les chemins existants. Le paramètre $mode peut prendre n'importe lesquelles des 3 valeurs suivantes : App : :APPEND, App : :PREPEND, `App::RESET`. Le behavior par défaut de la fonction reste le même (ex. Faire précéder des nouveaux chemins par une liste existante).

App : :path()
– Supporte maintenant les plugins, App : :path('Controller', 'Users') va retourner la localisation du dossier des controllers dans le plugin des users.
– Ne fusionnera plus les chemins du coeur, il retournera seulement les chemins définies dans App : :build() et ceux par défaut dans app (ou correspondant au plugin).

App : :build()
– Ne fusionnera plus le chemin de app avec les chemins du coeur.

App : :objects()
- Supporte maintenant les plugins, App : :objects('Users.Model') va retourner les models dans le plugin Users.
- Retourne array() au lieu de false pour les résultats vides ou les types invalides.
- Ne retourne plus les objets du coeur, App : :objects('core') retournera array().
- Retourne le nom complet de la classe.

La classe App perd les propriétés suivantes, utilisez la méthode App : :path() pour accéder à leur valeur
- App : :$models
- App : :$behaviors
- App : :$controllers
- App : :$components
- App : :$datasources
- App : :$libs
- App : :$views
- App : :$helpers
- App : :$plugins
- App : :$vendors
- App : :$locales
- App : :$shells

App : :import()
- Ne recherche plus les classes de façon récursive, il utilise strictement les valeurs pour les chemins définis dans App : :build().
- Ne sera plus capable de charger App : :import('Component', 'Component'), utilisez App : :uses('Component', 'Controller') ;
- Utiliser App : :import('Lib', 'CoreClass') pour charger les classes du coeur n'est plus possible.
- Importer un fichier non-existant, fournir un mauvais type ou un mauvais nom de package, ou des valeurs nulles pour les paramètres $name et $file va donner une fausse valeur de retour.
- App : :import('Core', 'CoreClass') n'est plus supporté, utilisez App : :uses() à la place et laisser la classe autoloading faire le reste.
- Charger des fichiers Vendor ne recherchera pas de façon récursive dans les dossiers Vendors, cela ne convertira plus le fichier en underscore comme cela se faisant dans le passé.

App : :core()
- Le premier paramètres n'est plus optionnel, il retournera toujours un chemin.
- Il ne peut plus être utilisé pour obtenir les chemins des vendors.
- Il acceptera seulement le nouveau style des noms de package.

Chargement des Classes avec App : :uses() Bien qu'il y ait eu une re-construction énorme dans la façon de charger les classes, pour quelques occasions, vous aurez besoin de changer le code de votre application pour respecter la façon que vous aviez l'habitude de faire. Le plus grand changement est l'introduction d'une nouvelle méthode :

```
App::uses('AuthComponent', 'Controller/Component');
```

Nous avons décidé que le nom de la fonction devait imiter le mot-clé `use` de PHP 5.3, juste pour la façon de déclarer où un nom de classe devait se trouver. Le premier paramètre de `App::uses()` est le nom complet de la classe que vous avez l'intention de charger, et le second paramètre, le nom du package (ou espace de noms) auquel il appartient. La principale différence avec le `App::import()` de CakePHP 1.3 est que l'actuelle n'importera pas la classe, elle configurera juste le système pour qu'à la première utilisation de la classe, elle soit localisée.

Quelques exemples de l'utilisation de `App::uses()` quand on migre de `App::import()` :

```
App::import('Controller', 'Pages');
// devient
App::uses('PagesController', 'Controller');

App::import('Component', 'Auth');
// devient
App::uses('AuthComponent', 'Controller/Component');

App::import('View', 'Media');
// devient
App::uses('MediaView', 'View');

App::import('Core', 'Xml');
// devient
App::uses('Xml', 'Utility');

App::import('Datasource', 'MongoDb.MongoDbSource')
// devient
App::uses('MongoDbSource', 'MongoDb.Model/Datasource')
```

Toutes les classes qui ont été chargées dans le passé utilisant `App::import('Core', $class);` auront besoin d'être chargées en utlisant `App::uses()` en référence au bon package. Voir l'api pour localiser les classes dans leurs nouveaux dossiers. Quelques exemples :

```
App::import('Core', 'CakeRoute');
// devient
App::uses('CakeRoute', 'Routing/Route');

App::import('Core', 'Sanitize');
// devient
App::uses('Sanitize', 'Utility');

App::import('Core', 'HttpSocket');
// devient
App::uses('HttpSocket', 'Network/Http');
```

Au contraire de la façon dont fonctionnait `App::import()`, la nouvelle classe de chargement ne va pas localiser les classes de façon récursive. Cela entraîne un gain de performance impressionnant même en mode développement, au prix de certaines fonctionnalités rarement utilisées qui ont toujours provoquées des effets secondaires. Pour être encore plus clair, la classe de chargement va seulement attraper la classe dans le package exact dans lequel vous lui avez dit de la trouver.

App : :build() et les chemins du coeur `App::build()` ne va plus fusionner les chemins de app avec les chemins du coeur.

Exemples :

```
App::build(array('controllers' => array('/chemin/complet/vers/controllers')))
//devient
App::build(array('Controller' => array('/chemin/complet/vers/controllers')))

App::build(array('helpers' => array('/chemin/complet/vers/controllers')))
//devient
App::build(array('View/Helper' => array('/chemin/complet/vers/Vues/Helpers')))
```

CakeLog

– La connexion aux flux a maintenant besoin de mettre en œuvre : php : class : *CakeLogInterface*. Des exceptions seront soulevées si un enregistreur n'est pas configuré.

Cache

– `Cache` est maintenant une classe statique, elle n'a plus de méthode getInstance().
– CacheEngine est maintenant une classe abstraite. Vous ne pouvez plus directement créer d'instances de celle-ci.
– Les implémentations de CacheEngine doivent étendre CacheEngine, des exceptions seront soulevées si une classe de configuration ne l'est pas.
– FileCache nécessite maintenant l'ajout de barres obliques au chemin de configuration lorsque vous modifiez une configuration du cache.
– Cache ne retient plus le nom du dernier moteur de cache configuré. Cela signifie que les opérations que vous souhaitez produire sur un moteur spécifique doivent avoir le paramètre $config égale au nom de config que vous souhaitez.

```
Cache::config('quelquechose');
Cache::write('key', $valeur);

// deviendrait
Cache::write('key', $valeur, 'quelquechose');
```

Router

– Vous ne pouvez plus modifier les paramètres de configuration avec `Router::setRequestInfo()`. Vous devriez utiliser `Router::connectNamed()` pour configurer la façon dont les paramètres nommés sont gérés.
– Le Router n'a plus de méthode `getInstance()`. C'est une classe statique, appelle ses méthodes et propriétés de façon statique.
– `Router::getNamedExpressions()` est deprécié. Utilisez les nouvelles constantes du routeur. `Router::ACTION`, `Router::YEAR`, `Router::MONTH`, `Router::DAY`, `Router::ID`, et `Router::UUID` à la place.

- `Router::defaults()` a été retiré. Supprimer l'inclusion de fichier des routes du coeur de votre fichier routes.php de vos applications pour désactiver le routing par défaut. Inversement, si vous voulez le routing par défaut, vous devrez ajouter une inclusion dans votre fichier de routes `Cake/Config/routes.php`.
- Quand vous utilisez Router : :parseExtensions() le paramètre d'extension n'est plus sous `$this->params['url']['ext']`. A la place, il est disponible avec `$this->request->params['ext']`.
- Les routes des plugins par défaut ont changé. Les routes courtes de Plugin ne sont plus construites que dans les actions index. Précédemment */users*' et `/users/add` mappaient le UsersController dans le plugin Users. Dans 2.0, seule l'action `index` est donné par une route courte. Si vous souhaitez continuer à utiliser les routes courtes, vous pouvez ajouter une route comme :

```
Router::connect('/users/:action', array('controller' => 'users', 'plugin' => 'users'));
```

  Pour votre fichier de routes pour chaque plugin, vous avez besoin de routes courtes actives.

Votre fichier app/Config/routes.php doit être mis à jour en ajoutant cette ligne en bas du fichier :

```
require CAKE . 'Config' . DS . 'routes.php';
```

Cela est nécessaire afin de générer les routes par défaut pour votre application. Si vous ne souhaitez pas avoir de telles routes, ou si vous voulez implémenter votre propre standard, vous pouvez inclure votre propre fichier avec vos propres règles de routeur.

Dispatcher

- Le Dispatcher a été déplacé dans cake/libs, vous devrez mettre à jour votre fichier `app/webroot/index.php`.
- Le `Dispatcher::dispatch()` prend maintenant deux paramètres. Les objets request et response. Ceux-ci devraient être des instances de `CakeRequest` & `CakeResponse` ou une sous-classe de ceux-ci.
- `Dispatcher::parseParams()` n'accepte que l'objet `CakeRequest`.
- `Dispatcher::baseUrl()` a été retiré.
- `Dispatcher::getUrl()` a été retiré.
- `Dispatcher::uri()` a été retiré.
- `Dispatcher::$here` a été retiré.

Configure

- `Configure::read()` avec aucun paramètre ne retourne plus la valeur de 'debug', à la place elle retourne toutes les valeurs dans Configure. Utilisez `Configure::read('debug');` si vous voulez la valeur de debug.
- `Configure::load()` requiert maintenant un ConfigReader pour être configuré. Lisez *Chargement des fichiers de configuration* pour plus d'informations.
- `Configure::store()` écrit maintenant les valeurs à une configuration du Cache donnée. Lisez *Chargement des fichiers de configuration* pour plus d'informations.

Scaffold

- Les vues Scaffold 'edit' devront être renommées par 'form'. Cela a été fait pour rendre les templates scaffold et bake cohérents.
  - `views/scaffolds/edit.ctp` -> `View/Scaffolds/form.ctp`
  - `views/posts/scaffold.edit.ctp` -> `View/Posts/scaffold.form.ctp`

Xml

- La classe Xml a été complètement reconstruite. Maintenant cette classe ne manipule plus de données, et elle est un enrouleur (wrapper) pour les SimpleXMLElement. Vous pouvez utiliser les méthodes suivantes :
  - `Xml::build()` : Méthode statique dans laquelle vous pouvez passer une chaîne de caractère xml, un tableau, un chemin vers un fichier ou une url. Le résultat va être une instance SimpleXMLElement ou une exception va être envoyée en cas d'erreurs.
  - `Xml::fromArray()` : Méthode statique qui retourne un SimpleXMLElement à partir d'un tableau.
  - `Xml::toArray()` : Méthode statique qui retourne un tableau à partir de SimpleXMLElement.

Vous devez utiliser la documentation `Xml` pour plus d'informations sur les changements faits sur la classe Xml.

Inflector

- L'Inflecteur n'a plus de méthode `getInstance()`.
- `Inflector::slug()` ne supporte plus l'argument $map. Utilisez `Inflector::rules()` pour définir les règles de translitération.

CakeSession

CakeSession est maintenant une classe complètement statique, les deux `SessionHelper` et `SessionComponent` sont des wrappers et du sucre pour celui-ci. Il peut facilement être utilisé dans les models ou dans d'autres contextes. Toutes ses méthodes sont appelées de façon statique.

La configuration de Session a aussi changé *Voir la section session pour plus d'informations*

HttpSocket

- HttpSocket ne change pas les clés d'en-tête. Suivant les autres endroits dans le coeur, le HttpSocket ne change pas les headers. **RFC 2616** (http ://tools.ietf.org/html/rfc2616.html) dit que les en-têtes sont insensibles à la casse, et HttpSocket préserve les valeurs envois de l'hôte distant.
- HttpSocket retourne maintenant les réponses en objets. Au lieu des tableaux, HttpSocket retourne les instances de HttpResponse. Voir la documentation de `HttpSocket` pour plus d'informations.
- Les cookies sont stockés en interne par l'hôte, pas par instance. Cela signifie que, si vous faîtes deux requêtes à différents serveurs, les cookies du domaine1 ne seront pas envoyés au domaine2. Cela a été fait pour éviter d'éventuels problèmes de sécurité.

Helpers

Changement du constructeur

Afin de prendre en considération le fait que View a été retiré de la ClassRegistry, la signature du Helper : :__construct() a été changée. Vous devez mettre à jour toutes les sous-classes pour utiliser ce qui suit :

```
public function __construct(View $View, $settings = array())
```

Quand vous écrasez le constructeur, vous devez toujours aussi appeler *parent : :__construct*. *Helper : :__construct* stocke l'instance de vue dans *$this->_View* pour une référence future. Les configurations ne sont pas gérées par le constructeur parent.

HelperCollection ajouté

Après un examen des responsabilités de chaque classe impliquée dans la couche Vue, il nous est clairement apparu que la Vue gérait bien plus qu'une unique tâche. La responsabilité de créer les helpers n'est pas centrale dans ce que la Vue fait, et a été déplacée dans le HelperCollection. HelperCollection est responsable du chargement et de la construction des helpers, ainsi que de déclencher les callbacks sur les helpers. Par défaut, la Vue crée un HelperCollection dans son constructeur, et l'utilise pour des opérations ultérieures. L'HelperCollection pour une vue peut être trouvé dans *$this->Helpers*.

Les motivations pour la reconstruction de cette fonctionnalité vient de quelques soucis.
- La Vue qui était enregistrée dans ClassRegistry pouvait causer des problèmes empoisonnés d'enregistrement quand requestAction ou l'EmailComponent étaient utilisés.
- La Vue accessible comme un symbole global entraînait des abus.
- Les Helpers n'étaient pas contenus eux-mêmes. Après avoir construit un helper, vous deviez construire manuellement plusieurs autres objets afin d'obtenir un objet fonctionnant.

Vous pouvez en lire plus sur HelperCollection dans la documentation *Collections*.

Propriétés dépréciées

Les propriétés suivantes sur les helpers sont depréciées, vous devez utiliser les propriétés de l'objet request ou les méthodes de l'Helper plutôt que accéder directement à ces propriétés puisqu'elles seront supprimées dans une version future.
- `Helper::$webroot` est depréciée, utilisez la propriété webroot de l'objet request.
- `Helper::$base` est depréciée, utilisez la propriété base de l'objet request.
- `Helper::$here` est depréciée, utilisez la propriété here de l'objet request.
- `Helper::$data` est depréciée, utilisez la propriété data de l'objet request.
- `Helper::$params` est depréciée, utilisez `$this->request` à la place.

XmlHelper, AjaxHelper et JavascriptHelper retirés

Les Helpers AjaxHelper et JavascriptHelper ont été retirés puisqu'ils étaient dépréciés dans la version 1.3. Le Helper XmlHelper a été retiré, puisqu'il était obsolète et superflu avec les améliorations de `Xml`. La

classe `Xml` doit être utilisée pour remplacer les utilisations anciennes de XmlHelper.

Les Helpers AjaxHelper et JavascriptHelper sont remplacés par les Helpers JsHelper et HtmlHelper.

JsHelper

– **`JsBaseEngineHelper` est maintenant abstrait, vous devrez implémenter** toutes les méthodes qui généraient avant des erreurs.

PaginatorHelper

– **`PaginatorHelper::sort()` prend maintenant les arguments title et key dans** l'ordre inversé. $key sera maintenant toujours le premier. Cela a été fait pour prévenir les besoins d'échange des arguments lors de l'ajout d'un second argument.
– PaginatorHelper avait un nombre de changements pour les paramètres de pagination utilisé en interne. Le key par défaut a été retiré.
– PaginatorHelper supporte maintenant la génération des liens avec les paramètres de pagination dans querystring.

Il y a eu quelques améliorations dans pagination en général. Pour plus d'informations sur cela, vous devriez lire la page des nouvelles fonctionnalités de pagination.

FormHelper

Le paramètre $selected retiré Le paramètre `$selected` a été retiré de plusieurs méthodes dans `FormHelper`. Toutes les méthodes supportent maintenant une clé `$attributes['value']` qui doit être utilisée à la place de `$selected`. Ce changement simplifie les méthodes `FormHelper`, réduit le nombre d'arguments, et réduit les répétitions que `$selected` créait. Les méthodes effectives sont :
– FormHelper : :select()
– FormHelper : :dateTime()
– FormHelper : :year()
– FormHelper : :month()
– FormHelper : :day()
– FormHelper : :hour()
– FormHelper : :minute()
– FormHelper : :meridian()

Les urls par défaut dans les formulaires sont l'action courante L'url par défaut pour tous les formulaires est maintenant l'url courante, incluant les paramètres passés, nommés et querystring. Vous pouvez écraser ce réglage par défaut en fournissant `$options['url']` dans le second paramètre de `$this->Form->create()`.

FormHelper : :hidden() Les champs cachés n'enlèvent plus la classe attribut. Cela signifie que si il y a des erreurs de validation sur des champs cachés, le nom de classe `error-field` sera appliqué.

CacheHelper

Le CacheHelper a été complètement découplé de la Vue, et des utilisations des callbacks du Helper pour générer des caches. Vous devez retenir de placer CacheHelper après les autres helpers qui modifient le contenu dans les callbacks `afterRender` et `afterLayout`. Si vous ne le faîtes pas, certains changements ne feront pas parti du contenu récupéré.

CacheHelper n'utilise également plus `<cake:nocache>` pour indiquer les régions non mises en cache. A la place, il utilise les commentaires spéciaux HTML/XML. `<!--nocache-->` et `<!--/nocache-->`. Cela aide CacheHelper à générer des balises valides et continue à effectuer les mêmes fonctions qu'avant. Vous pouvez en lire plus sur CacheHelper et les changements de Vue.

Les formats des attributs d'Helper plus flexibles

La classe Helper a 3 attributs protégés :
- `Helper::_minimizedAttributes` : tableau avec des attributs minimums (ex : `array('checked', 'selected', ...));`
- `Helper::_attributeFormat` : comment les attributs vont être générés (ex : `%s="%s"`);
- `Helper::_minimizedAttributeFormat` : comment les attributs minimums vont être générés : (ie `%s="%s"`)

Par défaut, les valeurs utilisées dans CakePHP 1.3 n'ont pas été changées. Mais vous pouvez maintenant utiliser les attributs boléens de HTML, comme `<input type="checkbox" checked />`. Pour cela, changez juste `$_minimizedAttributeFormat` dans votre AppHelper en `%s`.

Pour utiliser avec les helpers Html/Form et les autres, vous pouvez écrire :

```
$this->Form->checkbox('field', array('checked' => true, 'value' => 'une_valeur'));
```

Une autre aptitude est que les attributs minimums peuvent être passés en item et pas en clé. Par exemple :

```
$this->Form->checkbox('field', array('checked', 'value' => 'une_valeur'));
```

Notez que `checked` a une clé numérique.

Controller (Contrôleur)

- Le constructeur du Controller prend maintenant deux paramètres. Les objets CakeRequest et CakeResponse. Ces objets sont utilisés pour remplir plusieurs propriétés dépreciées et seront mis dans $request et $response à l'intérieur du controller.
- `Controller::$webroot` est depréciée, utilisez la propriété webroot de l'objet request.
- `Controller::$base` est depréciée, utilisez la propriété base de l'objet request.
- `Controller::$here` est depréciée, utilisez la propriété here de l'objet request.
- `Controller::$data` est depréciée, utilisez la propriété data de l'objet request.
- `Controller::$params` est depréciée, utilisez `$this->request` à la place.
- `Controller::$Component` a été déplacée vers `Controller::$Components`. Voir la documentation *Collections* pour plus d'informations.
- `Controller::$view` a été renommée en `Controller::$viewClass`. `Controller::$view` est maintenant utilisée pour changer le fichier vue qui doit être rendu.

– `Controller::render()` retourne maintenant un objet CakeResponse.

Les propriétés depréciées dans Controller seront accessibles à travers la méthode `__get()`. Cette méthode va être retirée dans les versions futures, donc il est recommandé que vous mettiez votre application à jour.

Le Controller définit maintenant une limite Max (maxLimit) pour la pagination. Cette limite maximale est mise à 100, mais peut être écrasée dans les options de $paginate.

Pagination

La Pagination était traditionnellement une unique méthode dans le Controller, cela créait pourtant un nombre de problèmes. La Pagination était difficile à étendre, remplacer et modifier. Dans 2.0, la pagination a été extraite dans un component. `Controller::paginate()` existe toujours, et sert en tant que méthode commode pour le chargement et en utilisant le `PaginatorComponent`.

Pour plus d'informations sur les nouvelles fonctionnalités offertes par la pagination dans 2.0, voir la documentation *Pagination*.

Vue

La Vue n'est plus enregistrée dans ClassRegistry

La vue enregistrée dans ClassRegistry entraînait des abus et créait effectivement un symbole global. Dans 2.0 chaque Helper reçoit l'instance *Vue* courante dans son constructeur. Cela autorise l'accès aux vues pour les helpers de la même façon que dans le passé, sans créer de symboles globaux. Vous pouvez accéder à l'instance de vue dans *$this->_View* dans n'importe quel helper.

Propriétés dépréciées

– `View::$webroot` est deprécié, utilisez la propriété webroot de l'objet request.
– `View::$base` est deprécié, utilisez la propriété base de l'objet request.
– `View::$here` est deprécié, utilisez la propriété here de l'objet request.
– `View::$data` est deprécié, utilisez la propriété data de l'objet request.
– `View::$params` est deprécié, utilisez `$this->request` à la place.
– `View::$loaded` a été retiré. Utilisez `HelperCollection` pour accéder aux helpers chargés.
– `View::$model` a été retiré. Ce behavior est maintenant dans `Helper`
– `View::$modelId` a été retiré. Ce behavior est maintenant dans `Helper`
– `View::$association` a été retiré. Ce behavior est maintenant dans `Helper`
– `View::$fieldSuffix` a été retiré. Ce behavior est maintenant dans `Helper`
– `View::entity()` a été retiré. Ce behavior est maintenant dans `Helper`
– `View::_loadHelpers()` a été retiré, utilisez `View::loadHelpers()` à la place.
– La façon dont `View::element()` utilise le cache a changé, voir en-dessous pour plus d'informations.
– Les callbacks de Vue ont été transférés, voir en-dessous pour plus d'informations.
– L'API pour `View::element()` a changé. Lire ici pour plus d'informations.

Les propriétés depréciées de Vue seront accessibles à travers une méthode `__get()`. Cette méthode va être retirée dans les versions futures, ainsi il est recommandé que vous mettiez à jour votre application.

Méthodes retirées

- `View::_triggerHelpers()` Utilisez `$this->Helpers->trigger()` à la place.
- `View::_loadHelpers()` Utilisez `$this->loadHelpers()` à la place. Les Helpers chargent maintenant facilement leurs propres helpers.

Méthodes ajoutées

- `View::loadHelper($name, $settings = array());` Charge un unique helper.
- `View::loadHelpers()` charge tous les helpers indiqués dans `View::$helpers`.

View->Helpers

Par défaut, les objets Vue contiennent un `HelperCollection` dans `$this->Helpers`.

Thèmes

Pour utiliser les thèmes dans vos Controllers, vous n'avez plus à mettre `var $view = 'Theme';`. Utilisez `public $viewClass = 'Theme';` à la place.

Changements de positionnement des callbacks

beforeLayout utilisé pour se déclencher après scripts_for_layout et content_for_layout a été préparé. Dans 2.0, beforeLayout est tiré avant que toute variable spéciale soit préparée, vous autorisant à les manipuler avant qu'elles soient passées au layout. La même chose a été faite pour beforeRender. Il est maintenant tiré bien avant que toute variable soit manipulée. En plus de ces changements, les callbacks des helpers reçoivent toujours le nom du fichier qui est sur le point d'être rendu. Ceci, combiné avec le fait que les helpers soient capables d'accéder à la vue à travers `$this->_View` et la vue courante du contenu à travers `$this->_View->output` vous donne plus de puissance qu'avant.

La signature du callback Helper change

Les callbacks de Helper récupèrent maintenant toujours un argument passé à l'intérieur. Pour BeforeRender et afterRender, c'est le fichier vue qui est rendu. Pour beforeLayout et afterLayout, c'est le fichier layout qui est rendu. Vos signatures de fonction des helpers doivent ressembler à cela :

```
public function beforeRender($viewFile) {

}

public function afterRender($viewFile) {

}
```

```
public function beforeLayout($layoutFile) {

}

public function afterLayout($layoutFile) {

}
```

L'élement attrapé, et les callbacks de vue ont été changés dans 2.0 pour vous aider à vous fournir plus de flexibilité et de cohérence. *Lire plus sur les changements*.

CacheHelper decouplé

Dans les versions precédentes, il y avait un couplage étroit entre `CacheHelper` et `View`. Dans 2.0 ce couplage a été retiré et CacheHelper utilise juste les callbacks comme les autres helpers pour générer la page complète mise en cache.

CacheHelper `<cake:nocache>` tags changés

Dans les versions précédentes, CacheHelper utilise un tag spécial `<cake:nocache>` comme marqueur pour la sortie qui ne devrait pas faire partie de la page entièrement mise en cache. Ces tags ne faisaient parti d'aucun schéma XML, et il n'était pas possible de valider dans les documents HTML et XML. Dans 2.0, ces tags ont été remplacés avec des commentaires HTML/XML :

```
<cake:nocache> devient <!--nocache-->
</cake:nocache> devient <!--/nocache-->
```

Le code interne pour la page vue complète mise en cache a aussi été changé, alors assurez vous de nettoyer le cache de la vue quand vous mettez à jour.

Changements de MediaView

`MediaView::render()` force maintenant le téléchargement de types de fichiers inconnus à la place de juste retourner false. Si vous le voulez, vous pouvez fournir un fichier de téléchargement alternatif, vous spécifiez le nom complet incluant l'extension en utilisant la clé 'name' dans le paramètre tableau passé à la fonction.

PHPUnit plutôt que SimpleTest

Tous les cas de test du coeur et les infrastructures supportant ont été portés pour utiliser PHPUnit 3.5. Bien sur, vous pouvez continuer à utiliser SimpleTest dans votre application en remplaçant les fichiers liés. Pas plus de support ne sera donné pour SimpleTest et il est recommandé que vous migriez vers PHPUnit aussi. Pour plus d'informations sur la façon de migrer vos tests, regardez les allusions sur la migration vers PHPUnit.

Plus de tests groupés

PHPUnit ne fait pas la différence entre les cas de tests groupés et les cas de tests uniques. A cause de cela, les options des tests groupés, et le support pour les tests groupés à l'ancienne ont été retirés. Il est recommandé que les TestGroupés soient portés vers les sous-classes de `PHPUnit_Framework_Testsuite`. Vous pouvez trouver plusieurs exemples de ceci dans la suite de test de CakePHP. Les méthodes liées aux tests groupés dans `TestManager` ont aussi été retirées.

Shell Testsuite

Le shell Testsuite a eu ses invocations simplifiées et étendues. Vous n'avez plus besoin de faire la différenciation entre `case` et `group`. On suppose que tous les tests sont des cas. Dans le passé, vous vous auriez fait `cake testsuite app case models/post`, vous pouvez maintenant faire `cake testsuite app Model/Post`.

Le shell Testsuite a été reconstruit pour utiliser l'outils cli de PHPUnit. Cela supporte maintenant toutes les options de ligne de commande supportées par PHPUnit. `cake testsuite help` vous montrera une liste de toutes les modifications possibles.

Models

Les relations des Models sont maintenant facilement chargées. Vous pouvez être dans une situation où l'assignation d'une valeur à une propriété non-existante d'un model vous enverra les erreurs :

```
$Post->inexistentProperty[] = 'value';
```

enverra à traver l'erreur "Notice : Indirect modification of overloaded property $inexistentProperty has no effect"(Notice : La modification indirecte d'une propriété $propriétéInexistente n'a aucun effet). Assigner une valeur initiale à la propriété résoud le problème :

```
$Post->nonexistentProperty = array();
$Post->nonexistentProperty[] = 'value';
```

Ou déclare juste la propriété dans la classe model :

```
class Post {
    public $nonexistentProperty = array();
}
```

Chacune des ses approches résoudra les erreurs de notice.

La notation de `find()` dans Cake 1.2 n'est plus supportée. Les Finds devront utiliser la notation `$model->find('type', array(PARAMS))` comme dans Cake 1.3.

- `Model::$_findMethods` est maintenant `Model::$findMethods`. Cette propriété est maintenant publique et peut être modifiée par les behaviors.

Objets Database (Base de Données)

Cake 2.0 introduit quelques changements dans les objets Database qui ne devraient pas affecter grandement la compatibilité rétro-active. Le plus grand changement est l'adoption de PDO pour la gestion des connections aux bases de données. Si vous utilisez une installation vanilla de PHP 5, vous aurez déjà les extensions nécessaires installées, mais il se peut que vous dussiez activer les extensions individuelles pour chaque driver que vous souhaitez utiliser.

Utiliser PDO à travers toutes les BDD nous permet d'homogénéiser le code pour chacune et fournit un comportement plus fiable et prévisible pour tous les drivers. Il nous a également permis d'écrire des tests plus précis et portables pour le code de la base de données liée.

La première chose qui va probablement manquer aux users, est les statistiques "lignes affectées" et "total de lignes", comme elles ne sont pas reportées à cause d'un design de PDO plus performant et paresseux, il y a des façons de régler ce problème, mais qui sont très spécifiques à chaque base de données. Ces statistiques ne sont pas parties cependant, mais pourraient manquer ou même être inexactes pour certains drivers.

Une fonctionnalité sympa ajoutée après l'adoption de PDO est la possibilité d'utiliser des requêtes préparées avec des placeholders de requêtes utilisant le driver natif si il est disponible.

Liste des changements

- DboMysqli a été retirée, nous ferons seulement le support de DboMysql.
- API pour DboSource : :execute a changé, elle prendra maintenant un tableau de valeurs requêtées en second paramètre :

```
public function execute($sql, $params = array(), $options = array())
```

  devient :

```
public function execute($sql, $options = array(), $params = array())
```

  le troisième paramètre est supposé recevoir les options pour se connecter, en ce moment, il ne comprend que l'option "log".
- DboSource : :value() perd son troisième paramètre, il n'était pas utilisé de toute façon.
- DboSource : :fetchAll() accepte maintenant un tableau en second paramètre, pour passer les valeurs devant être liées à la requête, le troisième paramètre a été abandonnée. Exemple :

```
$db->fetchAll('SELECT * from users where nom_utilisateur = ? AND mot_de_passe = ?', array
$db->fetchAll('SELECT * from users where nom_utilisateur = :nom_utilisateur AND mot_de_pa
```

Le driver PDO va automatiquement echapper ces valeurs pour vous.

- Les statistiques de Base de données sont collectées seulement si la propriété "fullDebug" de la BDD correspondante est mise à true.
- Nouvelle méthode DboSource : :getConnection() va retourner l'objet PDO dans le cas où vous auriez besoin de parler directement au driver.
- Le traitement des valeurs boléennes a changé un peu pour pouvoir faciliter le croisement de base de données, vous devrez peut-être changer vos cas de test.
- Le support de Postgresql a été immensément amélioré, il crée maintenant correctement les schémas, vide les tables, et il est plus facile d'écrire des tests en l'utilisant.
- DboSource : :insertMulti() n'acceptera plus les chaînes sql, passez juste un tableau de champs et un tableau imbriqué de valeurs pour les insérer tous en une fois.

- TranslateBehavior a été reconstruit pour utiliser les vituaFields des models, cela rend l'implémentation plus portable.
- Tous les cas de test avec les choses liées de Mysql ont été déplacés vers le cas de test du driver correspondant. Cela a laissé le fichier DboSourceTest un peu maigre.
- Support de l'imbrication des transactions. Maintenant il est possible de démarrer une transaction plusieurs fois. Il ne peut être engagé si la méthode de validation est appelé le même nombre de fois.
- Le support Sqlite a été grandement amélioré. La différence majeure avec cake 1.3 est qu'il ne supportera que Sqlite 3.x. C'est une bonne alternative pour le développement des apps, et rapidement en lançant les cas de test.
- Les valeurs des colonnes boléennes vont être lancées automatiquement vers le type boléeen natif de php, donc assurez vous de mettre à jour vos cas de test et code si vous attendiez une valeur retournée de type chaîne de caractère ou un entier : Si vous aviez une colonne "published" dans le passé en utilisant mysql, toutes les valeurs retournées d'un find auraient été numériques dans le passé, maintenant elles sont strictement des valeurs boléennes.

BehaviorCollection

- `BehaviorCollection` ne met plus en minuscule `strtolower()` les mappedMethods. Les mappedMethods des Behaviors sont maintenant sensible à la casse.

AclBehavior et TreeBehavior

- Ne supporte plus les chaînes de caractère pour la configuration. Exemple :

```
public $actsAs = array(
    'Acl' => 'Controlled',
    'Tree' => 'nested'
);
```

devient :

```
public $actsAs = array(
    'Acl' => array('type' => 'Controlled'),
    'Tree' => array('type' => 'nested')
);
```

Plugins

Les plugins n'ajoutent plus de façon magique leur prefix plugin aux components, helpers et models utilisés à travers eux. Vous devez être explicites avec les components, models et helpers que vous souhaitez utiliser. Dans le passé :

```
var $components = array('Session', 'Comments');
```

Aurait regardé dans le plugin du controller avant de vérifier les components app/core. Il va maintenant seulement regarder dans les components app/core. Si vous souhaitez utiliser les objets à partir d'un plugin, vous devez mettre le nom du plugin :

```
public $components = array('Session', 'Comment.Commentaires');
```

Cela a été fait pour réduire la difficulté des problèmes de debug causés par les ratés de la magie. Cela améliore aussi la cohérence dans votre application, puisque les objets ont une façon autoritaire de les référencer.

Plugin App Controller et Plugin App Model

Les plugins AppController et AppModel ne sont plus directement localisés dans le dossier plugin. Ils sont maintenant placés dans les dossiers des plugins des Controllers et des Models comme ceci :

```
/app
    /Plugin
        /Comment
            /Controller
                CommentAppController.php
            /Model
                CommentAppModel.php
```

Console

La plupart de la console du framework a été reconstruite pour 2.0 pour traiter un grand nombre de questions suivantes :
- Etroitement couplé.
- Il était difficile de faire un texte d'aide pour les shells.
- Les paramètres pour les shells étaient fastidieux à valider.
- Les tâches des Plugins n'étaient pas joignables.
- Objets avec trop de responsabilités.

Backwards incompatible Shell API changes

- `Shell` n'a plus d'instance `AppModel`. Cette instance `AppModel` n'était pas correctement construite et était problématique.
- `Shell::_loadDbConfig()` a été retiré. Il n'était pas assez générique pour rester dans le Shell. Vous pouvez utiliser `DbConfigTask` si vous avez besoin de demander à l'utilisateur de créer une config db.
- Shells n'utilise plus `$this->Dispatcher` pour accéder à stdin, stdout, et stderr. Ils ont maintenant les objets `ConsoleOutput` et `ConsoleInput` pour gérer cela.
- Shells chargent les tâches facilement, et utilisent `TaskCollection` pour fournir une interface similaire à celle utilisée pour les Helpers, Components, et Behaviors pour le chargement à la volée des tâches.
- `Shell::$shell` a été retiré.
- `Shell::_checkArgs()` a été retiré. Configurer un `ConsoleOptionParser`
- Shells n'ont plus d'accès direct à `ShellDispatcher`. Vous devez utiliser les objets `ConsoleInput` et `ConsoleOutput` à la place. Si vous avez besoin de dispatcher d'autres shells, regardez la section sur 'invoquer d'autres shells à partir de votre shell'.

Backwards incompatible ShellDispatcher API changes

- `ShellDispatcher` n'a plus de fichiers de gestion stdout, stdin, stderr.
- `ShellDispatcher::$shell` a été retirée.
- `ShellDispatcher::$shellClass` a été retirée.
- `ShellDispatcher::$shellName` a été retirée.
- `ShellDispatcher::$shellCommand` a été retirée.
- `ShellDispatcher::$shellPaths` a été retirée, utilisez `App::path('shells');` à la place.
- `ShellDispatcher` n'utilise plus 'help' comme méthode magique qui a un statut spécial. A la place, utilisez les options `--help/-h`, et un parseur d'option.

Backwards incompatible Shell Changes

- Bake's ControllerTask ne prend plus `public` et `admin` comme arguments passés. Ce sont maintenant des options, indiquées par `--admin` et `--public`.

Il est recommandé que vous utilisiez le help sur les shells que vous utilisiez pour voir si tous les paramètres ont changé. il est aussi recommandé que vous lisiez les nouvelles fonctionnalités de la console, pour plus d'informations sur les nouvelles APIs qui sont disponibles.

Debugger

La fonction `debug()` va sortir par défaut les chaînes sans danger de html. C'est désactivé si c'est utilisé dans la console. L'option `$showHtml` pour `debug()` peut être mis sur false pour désactiver la sortie sans danger de html du debug.

ConnectionManager

`ConnectionManager::enumConnectionObjects()` va maintenant retourner la configuration courante pour chaque connection créée, au lieu d'un tableau avec un nom de fichier, d'un nom de classe et d'un plugin, qui n'était pas réellement utiles.

Quand vous définirez les connections à la base de données, vous aurez besoin de faire quelques changements dans la façon dont les configs ont été définies dans le passé. Basiquement dans classe de configuration de la base de données, la clé "driver" n'est plus acceptée, seulement "datasource", afin de la rendre plus cohérente. Aussi, comme les sources de données ont été déplacées vers les packages, vous aurez besoin de passer le package dans lequel ils sont localisés. Exemple :

```
public $default = array(
    'datasource' => 'Database/Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'login' => 'root',
    'password' => 'root',
    'database' => 'cake',
);
```

Nouvelles caractéristiques dans CakePHP 2.0

Model

Le processus de construction du modèle a été allegé. Les associations des Modèles sont maintenant en lazy loaded, les applications avec beaucoup de modèles et d'associations vont voir une grande réduction de temps dans le processus de bootstrap.

Les modèles ne requièrent maintenant plus de connection à la base de données dans le processus de construction. La base de données ne sera accédée pour la première fois seulement quand une opération de recherche est délivrée ou une information pour une des colonnes est requise.

View

View : :$output

View n'aura pas toujours le dernier rendu de contenu (view ou layout) accessible avec `$this->output`. Dans les helpers, vous pouvez utiliser `$this->_View->output`. Modifier cette propriété changera le contenu qui sort de la vue rendue.

Helpers

HtmlHelper

- `getCrumbList()` Créé un fil d'ariane de liens entourés d'éléments `<li>`. Regardez #856 (http ://cakephp.lighthouseapp.com/projects/42648/tickets/856).
- `loadConfig()` a été déplacé de `Helper` vers la classe `HtmlHelper`. Cette méthode utilise maintenant les nouvelles classes de lecture (voir 2.0 `Configure`) pour charger votre fichier de config. En option vous pouvez passer le chemin en deuxième paramètre (`app/Config` par défaut). Pour simplifier, vous pouvez définir le fichier de configuration (et le lecteur) dans `Controller::$helpers` (exemple ci-dessous) pour charger le constructeur du helper. Dans le fichier de configuration, vous pouvez définir les clés ci-dessous :

- `tags` Doit être un tableau avec une valeur clé ;
- `minimizedAttributes` Doit être une liste ;
- `docTypes` Doit être un tableau avec une valeur clé ;
- `attributeFormat` Doit être une chaîne de caractère ;
- `minimizedAttributeFormat` Doit être une chaîne de caractère.

Exemple sur la façon de définir le fichier de configuration sur les contrôleurs :

```
public $helpers = array(
    'Html' => array(
        'configFile' => array('config_file', 'php') // Option une: un tableau avec le nom d
        'configFile' => 'config_file' // Option deux: une chaîne de caractère avec le nom d
    )
);
```

FormHelper

- `FormHelper` supporte maintenant tout type d'entrée HTML5 et tout type d'entrée personnalisé. Utilisez simplement le type d'entrée que vous souhaitez en méthode sur le helper. Par exemple `range()` créera une entrée avec type = range.
- `postLink()` et `postButton()` Crée un lien/bouton permettant d'accéder à certaine pasge utilisant la méthode HTTP POST. Avec ceci dans votre controller vous pouvez empêcher certaines actions, comme delete, d'être accédées par la méthode GET.
- `select()` avec multiple = checkbox, traite maintenant l'attribut `'id'` en préfixe pour toutes les options générées.

Libs

CakeRequest

`CakeRequest` est une nouvelle classe introduite dans 2.0. Elle encapsule les méthodes d'introspection de requêtes utilisées couramment et remplace le tableau params avec un objet plus utile. Lisez en plus sur `CakeRequest`.

CakeResponse

`CakeResponse` est une nouvelle classe introduite dans 2.0. Elle encapsule les méthodes et propriétés utilisées couramment dans la réponse HTTP que votre application génére. Elle consolide plusieurs caractéristiques dans CakePHP. Lisez en plus sur `CakeResponse`.

CakeSession, SessionComponent

`CakeSession` et le `SessionComponent` ont connu un nombre de changements, regardez la section session pour plus d'informations.

Router

Routes peuvent retourner des urls complètes Les Objets Route peuvent maintenant retourner des urls complètes, et `Router` ne les modifiera plus au-delà de l'ajout de la chaîne de requête et des éléments de fragments. Par exemple, ceci pouvait être utilisé pour créer des routes pour la gestion de sous-domaines, ou pour l'activation de flags https/http. Un exemple de classe de route qui supporte les sous-domaines serait :

```
class SubdomainRoute extends CakeRoute {

    public function match ($params) {
        $subdomain = isset($params['subdomain']) ? $params['subdomain'] : null;
        unset($params['subdomain']);
        $path = parent::match($params);
        if ($subdomain) {
            $path = 'http://' . $subdomain . '.localhost' . $path;
```

```
        }
        return $path;
    }
}
```

Quand vous créez des liens, vous pouvez faire ce qui suit pour faire pointer les liens vers d'autres sous-domaines.

```
echo $this->Html->link(
    'Autre domaine',
     array('subdomain' => 'test', 'controller' => 'posts', 'action' => 'add')
);
```

Ce qui est ci-dessus créera un lien avec l'url http ://test.localhost/posts/add.

Xml

`Xml` a connu un certain nombre de changements. Lisez en plus sur la classe *Xml*.

Nouvelles caractéristiques de Lib

Configure readers

`Configure` peut maintenant être configuré pour le chargement de fichiers à partir d'une variété de sources et de formats. La section *Configuration* contient plus d'informations sur les changements faits à configure.

`Configure::read()` sans autre argument vous permet de lire toutes les valeurs de configure, plutôt que uniquement la valeur du debug.

Error et gestion des exceptions

CakePHP 2.0 a reconstruit la gestion des *Exceptions* et des *Gestion des Erreurs*, pour être plus flexible et donner plus de puissance aux développeurs.

String : :wrap()

`String::wrap()` a été ajouté pour faciliter les formatages de largeur fixe des textes. Il est utilisé dans les Shells quand vous utilisez `Shell::wrapText()`.

debug()

`debug()` ne sort plus de html dans la console. A la place, elle donne des sorties comme ce qui suit :

```
########## DEBUG ##########
Array
(
    [0] => test
)
###########################
```

Ceci devrait améliorer la lecture de `debug()` dans les lignes de commande.

Components

Components reçoit un traitement identique aux helpers et aux behaviors, `Component` est maintenant la classe de base pour les components. Lisez en plus sur les changements sur les components.

RequestHandler

`RequestHandler` a été fortement remaniée du fait de l'introduction de `CakeRequest`. Ces changements permettent à certaines nouvelles fonctionnalités d'être aussi introduites.

Parsing automatique d'Acceptation des headers Si un client envoie un unique mime type Accept qui correspond à l'une des extensions activées dans :php :class'Router', `RequestHandler` le traitera de la même façon qu'une extension. Cela étendra le support de CakePHP pour les terminaux de type REST. Pour utiliser cette fonctionnalité, commencez par activer les extensions dans `app/Config/routes.php`

```
Router::parseExtensions('json', 'xml');
```

Une fois que vous avez créé les layouts et les vues pour vos extensions, vous pourrez visiter une url comme posts/view/1 et envoyer Accept : `application/json` dans les headers pour recevoir la version json de cette url.

CookieComponent

`CookieComponent` supporte maintenant seulement les cookies http. Vous pouvez les activer en utilisant `$this->Cookie->httpOnly = true;`. Avoir seulement les cookies http les rendra inaccessible à partir du navigateur.

Security Component CSRF separation

CakePHP a une protection CSRF depuis 1.2. Pour 2.0, le CSRF existant a un nouveau mode plus paranoïaque, et est sa caractéristique propre autonome. Dans le passé, les fonctionnalités CSRF étaient couplées avec des gardes-fous de tampering de formulaires. Les développeurs désactivent souvent validatePost pour faire des formulaires dynamiques, en désactivant la protection CSRF en même temps. Pour 2.0, la vérification CSRF a été séparée du tampering des formulaires vous donnant plus de contrôle.

Pour plus d'informations, regardez *protection CSRF*

Controller

Les Controllers ont maintenant accès aux objets request et response. Vous pouvez en lire plus sur ces objets sur leurs pages spécifiques.

Console

La console pour CakePHP 2.0 a été preque entièrement reconstruite. De nombreuses nouvelles caractéristiques ainsi que quelques changements incompatibles avec antérieurement. Lisez en plus sur les changements sur la console.

Pagination

Pagination fournit maintenant un maxLimit par défaut à 100 pour la pagination.

Cette limite peut maintenant être dépassée avec la variable paginate dans le Controller.

```
$this->paginate = array('maxLimit' => 1000);
```

Cette valeur par défaut est fournie pour empêcher l'utilisateur de manipuler les URL provoquant une pression excessive sur la base de données pour les requêtes suivantes, où un utilisateur modifierait le paramètre 'limit' pour une nombre très important.

Mettre un Alias

Vous pouvez maintenant mettre un alias les helpers, les components et les behaviors pour utiliser votre classe plutôt qu'une autre. Cela signifie que vous pouvez très facilement faire un helper `MyHtml` et n'avez pas besoin de remplacer chaque instance de `$this->Html` dans vos vues. Pour le faire, passez la clé 'className' tout au long de votre classe, comme vous feriez avec les modèles.

```
public $helpers = array(
    'Html' => array(
        'className' => 'MyHtml'
    )
);
```

De même, vous pouvez mettre en alias les components pour l'utilisation dans vos controllers.

```
public $components = array(
    'Email' => array(
        'className' => 'QueueEmailer'
    )
);
```

Appeller le component Email appelle le component QueueEmailer à la place. Finalement, vous pouvez aussi mettre en alias les behaviors.

```
public $actsAs = array(
    'Containable' => array(
        'className' => 'SuperContainable'
    )
);
```

Du fait de la façon dont 2.0 utilise les collections et les partage dans toute l'application, toute classe que vous mettez en alias sera utilisée dans toute votre application. Quelque soit le moment où votre application essaie d'accéder à l'alias, elle aura accès à votre classe. Par exemple, quand vous mettez en alias le helper Html dans l'exemple ci-dessus, tous les helpers qui utilisent le helper Html ou les éléments qui chargent le helper Html, utiliseront MyHtml à la place.

ConnectionManager

Une nouvelle méthode `ConnectionManager::drop()` a été ajoutée pour permettre de retirer les connections lors de l'éxecution.

PHPUnit Migration Hints

Migrating your test cases to PHPUnit 3.5 (http ://www.phpunit.de/manual/current/en/) will hopefully be a fairly pain free transition. However, there are a few known differences between test cases under PHPUnit and SimpleTest (http ://www.simpletest.org/).

Installing PHPUnit

Installing from PEAR (Recommended)

PHPUnit recommends it be installed via the pear installer, to do so run the following commands :

```
pear channel-discover pear.phpunit.de
pear channel-discover components.ez.no
pear channel-discover pear.symfony-project.com
```

This has to be done only once. Now the PEAR Installer (http ://pear.php.net/) can be used to install packages from the PHPUnit channel :

```
pear install phpunit/PHPUnit
```

Installing manually

In addition to pear, CakePHP also supports placing the PHPUnit directory inside one of your vendors directories along with all its dependencies. Doing so is generally not recommended as PHPUnit is complicated to install as it is composed of many pear packages. Installing with the pear installer is easier and faster.

If you do wish to install PHPUnit manually, you'll need to place it and all of its dependencies inside your applications `Vendor` directory.

Differences between SimpleTest

There are a number of differences between SimpleTest and PHPUnit. The following is an attempt to list the most frequently encountered differences.

startCase() and endCase()

These methods are no longer supported. Use the static methods PHPUnit provides : `setupBeforeClass` and `tearDownAfterClass`.

start(), end(), before() and after()

These methods were part of SimpleTest's test case initialization. `start()` and `end()` have no replacements. You can use `setUp()` and `tearDown()` to replace `before()` and `after()`.

setUp() and tearDown()

In the past the methods `setUp`, `tearDown`, `startTest` and `endTest` where supported, and caused confusion as they looked almost like the same thing but in some cases you should use one or the other.

In the new CakePHP test suite, it is recommended to use only `setUp` and `tearDown`. The methods startTest and endTest are still supported but are deprecated.

getTests

The method `getTests` is no longer supported. You can use filters instead. The web test runner now takes an additional query string parameter that allows you to specify a basic regular expression. This regular expression is used to restrict the methods that are run :

```
e.g. filter=myMethod
```

Only tests containing the string `myMethod` will be run on the next refresh. The cake testsuite shell also supports a -filter option to filter methods.

Assertion methods

Many of the assertion methods have slightly different names between PHPUnit and SimpleTest. Where possible `CakeTestCase` provides a wrapper for the SimpleTest method names. These compatibility wrappers will be removed in 2.1.0. The following methods will be affected.
– `assertEqual` -> `assertEquals`
– `assertNotEqual` -> `assertNotEquals`
– `assertPattern` -> `assertRegExp`
– `assertIdentical` -> `assertSame`
– `assertNotIdentical` -> `assertNotSame`
– `assertNoPattern` -> `assertNotRegExp`

- `assertNoErrors` -> no replacement
- `expectError` -> `setExpectedException`
- `expectException` -> `setExpectedException`
- `assertReference` -> `assertSame`
- `assertIsA` -> `assertType`

Some methods take their arguments in different orders, be sure to check the methods you are using when updating them.

Mock expectations

Mock objects are dramatically different between PHPUnit and SimpleTest. There is no compatibility wrapper between them. Updating mock object usage can be a painful process but we hope the following tips help you in your migration. It's highly recommended you familiarize yourself with the PHPUnit Mock object (http ://www.phpunit.de/manual/current/en/test-doubles.html#test-doubles.mock-objects) documentation.

Replacing method calls The following regular expressions should help update some of your more straightforward mock object expectations.

Replace expectOnce() no params

```
expectOnce\(([^\)]+)\);
expects(\$this->once())->method($1);
```

Replace expectOnce() with params

```
expectOnce\(([^,]+), array\((.+)\)\);
expects(\$this->once())->method($1)->with($2);
```

Replace expectAt()

```
expectAt\((\d+), (.+), array\((.+)\)\);
expects(\$this->at($1))->method($2)->with($3);
```

Replace expectNever

```
expectNever\(([^\)]+)\);
expects(\$this->never())->method($1);
```

Replace setReturnValue

```
setReturnValue\(([^,]+), (.+)\);
expects(\$this->once())->method($1)->will($this->returnValue($2));
```

Replace setReturnValueAt

```
setReturnValueAt((\d+), ([^,]+), (.+));
expects(\$this->at($1))->method($2)->will($this->returnValue($3));
```

Group tests

Group tests have been removed as PHPUnit treats individual test cases and test suites as composable entities in the runner. You can place group tests inside the cases directory and use `PHPUnit_Framework_TestSuite` as a base class. An example Testsuite would look like :

```
class AllJavascriptHelpersTest extends PHPUnit_Framework_TestSuite {

/**
 * Suite define the tests for this suite
 *
 * @return void
 */
    public static function suite() {
        $suite = new PHPUnit_Framework_TestSuite('Js Helper and all Engine Helpers');

        $helperTestPath = CORE_TEST_CASES . DS . 'View' . DS . 'Helper' . DS;
        $suite->addTestFile($helperTestPath . 'JsHelperTest.php');
        $suite->addTestFile($helperTestPath . 'JqueryEngineHelperTest.php');
        $suite->addTestFile($helperTestPath . 'MootoolsEngineHelperTest.php');
        $suite->addTestFile($helperTestPath . 'PrototypeEngineHelperTest.php');
        return $suite;
    }
}
```

`TestManger` no longer has methods to add tests to group tests either. It is recommended that you use the methods PHPUnit offers.

Migration de la version 1.2 vers la 1.3

Migrer de CakePHP 1.2 vers 1.3

Ce guide résume plusieurs des changements nécessaires quand on migre du coeur de Cake 1.2 vers 1.3. Chaque section contient des informations pertinentes pour les modifications faîtes aux méthodes existantes ainsi que toute méthode qui a été retirée/renommée.

Remplacements du fichier App (important)
- webroot/index.php : Doit être remplacé à cause des changements dans le processus de bootstrapping.
- config/core.php : Des configurations additionnelles ont été mise en place qui sont requises pour PHP 5.3.
- webroot/test.php : Remplacez si vous voulez lancer des tests unitiaires.

Constantes retirées

Les constantes suivantes ont été retirées de CakePHP. Si votre application dépend d'eux, vous devez les définir dans `app/config/bootstrap.php`

- `CIPHER_SEED` - Cela a été remplacé par la variable `Security.cipherSeed` de la classe de configuration qui doit être changée dans `app/config/core.php`
- `PEAR`
- `INFLECTIONS`
- `VALID_NOT_EMPTY`
- `VALID_EMAIL`
- `VALID_NUMBER`
- `VALID_YEAR`

Configuration et bootstrapping de l'application

Chemins de bootstrapping en plus.

Dans votre fichier app/config/bootstrap.php il se peut que vous ayez des variables telles que `$pluginPaths` ou `$controllerPaths`. Il y a une nouvelle façon d'ajouter ces chemins. Comme dans la 1.3 RC1, les variables `$pluginPaths` ne fonctionneront plus. Vous devez utiliser `App::build()` pour modifier les chemins.

```
App::build(array(
    'plugins' => array('/chemin/complet/vers/plugins/', '/prochain/chemin/complet/vers/plu
    'models' =>  array('/chemin/complet/vers/models/', '/prochain/chemin/complet/vers/model
    'views' => array('/chemin/complet/vers/vues/', '/prochain/chemin/complet/vers/vues/'),
    'controllers' => array('/chemin/complet/vers/controllers/', '/prochain/chemin/complet/v
    'datasources' => array('/chemin/complet/vers/sources_de_données/', '/prochain/chemin/co
    'behaviors' => array('/chemin/complet/vers/behaviors/', '/prochain/chemin/complet/vers/
    'components' => array('/chemin/complet/vers/components/', '/prochain/chemin/complet/ve
    'helpers' => array('/chemin/complet/vers/helpers/', '/prochain/chemin/complet/vers/help
    'vendors' => array('/chemin/complet/vers/vendors/', '/prochain/chemin/complet/vers/vend
    'shells' => array('/chemin/complet/vers/shells/', '/prochain/chemin/complet/vers/shells
    'locales' => array('/chemin/complet/vers/locale/', '/prochain/chemin/complet/vers/local
    'libs' => array('/chemin/complet/vers/libs/', '/prochain/chemin/complet/vers/libs/')
));
```

Ce qui a aussi changé est l'ordre dans lequel apparait le bootstrapping. Dans le passé, `app/config/core.php` était chargé **après** `app/config/bootstrap.php`. Cela entraînait que n'importe quel `App::import()` dans le bootstrap d'une application n'était plus en cache et ralentissait considérablement par rapport à une inclusion en cache. Dans 1.3, le fichier core.php est chargé et les configurations du coeur mises en cache sont créées **avant** que bootstrap.php soit chargé.

Chargement des inflections

`inflections.php` a été retiré, c'était un fichier non nécessaire et les fonctionnalités liées ont été reconstruites dans une méthode pour augmenter leur flexibilité. Vous pouvez maintenant utiliser `Inflector::rules()` pour charger les différentes inflections.

```
Inflector::rules('singular', array(
    'rules' => array('/^(bil)er$/i' => '\1', '/^(inflec|contribu)tors$/i' => '\1ta'),
    'uninflected' => array('singulars'),
    'irregular' => array('spins' => 'spinor')
));
```

Fusionnera les règles fournies dans un ensemble d'inflections, avec les règles ajoutées prenant le pas sur les règles de base.

Renommages de fichier et changements internes

Renommage des Librairies

Les librairies du coeur de libs/session.php, libs/socket.php, libs/model/schema.php et libs/model/behavior.php ont été renommées afin qu'il y ait une meilleure correspondance entre les noms de fichiers et les principales classes contenues (ainsi que la gestion avec les problèmes d'espaces de noms) :

- session.php -> cake_session.php
  - App : :import('Core', 'Session') -> App : :import('Core', 'CakeSession')
- socket.php -> cake_socket.php
  - App : :import('Core', 'Socket') -> App : :import('Core', 'CakeSocket')
- schema.php -> cake_schema.php
  - App : :import('Model', 'Schema') -> App : :import('Model', 'CakeSchema')
- behavior.php -> model_behavior.php
  - App : :import('Core', 'Behavior') -> App : :import('Core', 'ModelBehavior')

Dans la plupart des cas, le renommage ci-dessus, n'affectera pas les codes existants.

Héritage de Object

Les classes suivantes ne vont plus étendre Object :

- Router
- Set
- Inflector
- Cache
- CacheEngine

Si vous utilisiez les méthodes de Object à partir de ces classes, vous devrez ne plus utiliser ces méthodes.

Controllers & Components

Controller

- `Controller::set()` ne change plus les variables à partir de `$var_name` vers `$varName`. Les variables apparaissent toujours dans la vue exactement comme vous l'aviez fixée.
- `Controller::set('title', $var)` ne fixe plus `$title_for_layout` quand il rend le layout. `$title_for_layout` est toujours rempli par défaut. Mais si vous voulez le modifier, utilisez `$this->set('title_for_layout', $var)`.
- `Controller::$pageTitle` a été retiré. Utilisez `$this->set('title_for_layout', $var);` à la place.

– Controller a deux nouvelles méthodes `startupProcess` et `shutdownProcess`. Ces méthodes sont responsables de la gestion du startup du controller et des processus de shutdown.

Component

– `Component::triggerCallback` a été ajouté. C'est un hook générique dans le processus de callback du component. Il supplante `Component::startup()`, `Component::shutdown()` et `Component::beforeRender()` comme manière préférentielle pour déclencher les callbacks.

CookieComponent

– `del` est dépreciée, utilisez `delete`

AclComponent + DbAcl

La vérification de la référence du Noeud faite avec les chemins sont maintenant moins gourmands et ne consommeront plus les noeuds intermédiaires quand on fait des recherches. Dans le passé, étant donné la structure :

```
ROOT/
    Users/
          Users/
                edit
```

Le chemin `ROOT/Users` correspondrait au dernier noeud Users au lieu du premier. Dans 1.3, si vous vous attenidez à obtenir le dernier noeud, vous deviez utiliser le chemin `ROOT/Users/Users`

RequestHandlerComponent

– `getReferrer` est déprecié, utilisez `getReferer`

SessionComponent & SessionHelper

– `del` est déprecié, utilisez `delete`

`SessionComponent::setFlash()` Le second paramètre utilisé habituellement pour configurer le layout et par conséquence le rendu du fichier layout. Cela a été modifié pour utiliser un élément. Si vous spécifiez des flash de session dans vos applications vous aurez besoin de faire les changements suivants.

1. Déplacer les fichiers de layout requis dans app/views/elements
2. Renommer la variable $content_for_layout en $message
3. Assurez vous d'avoir `echo $session->flash();` dans votre layout

`SessionComponent` et `SessionHelper` ne sont pas chargés automatiquement. Les deux helpers `SessionComponent` et `SessionHelper` ne sont plus inclus automatiquement sans que vous le demandiez. SessionHelper et SessionComponent se comportent maintenant comme chaque autre component et doivent être déclarés comme tout autre helper/component. Vous devriez mettre à jour `AppController::$components` et `AppController::$helpers` pour inclure ces classes pour conserver les behaviors existants.

```
var $components = array('Session', 'Auth', ...);
var $helpers = array('Session', 'Html', 'Form' ...);
```

Ces changements ont été faits pour rendre CakePHP plus explicites et déclaratifs dans quelles classes, vous le développeur d'applications, veut l'utiliser. Dans le passé, il n'y avait aucun moyen d'éviter le chargement des classes de Session sans modifier les fichiers du coeur. Ce qui est quelque chose que nous souhaitions que vous soyez capable d'éviter. De plus, les classes de Session étaient le seul component ou helper magique. Ce changement aide à unifier et normaliser le behavior pour toutes les classes.

Classes de Librairie

CakeSession
– `del` est déprecié, utilisez `delete`
SessionComponent
– `SessionComponent::setFlash()` utilise maintenant un *élément* au lieu d'un *layout* en second paramètre. Assurez vous de déplacer tout flash layout de app/views/layouts vers app/views/elements et de changer les instances de $content_for_layout en $message.
Folder
– `mkdir` est déprecié, utilisez `create`
– `mv` est déprecié, utilisez `move`
– `ls` est déprecié, utilisez `read`
– `cp` est déprecié, utilisez `copy`
– `rm` est déprecié, utilisez `delete`
Set
– `isEqual` est déprecié, utilisez == ou ===.
String
– `getInstance` est déprecié, appelez les méthodes String statiquement.
Router

`Routing.admin` est déprecié. Il fournit un behavior incompatible avec les autres styles de prefix de routes puisqu'il était traité différemment. A la place, vous devez utiliser `Routing.prefixes`. Les préfixes de routes dans 1.3 ne nécessitent pas la déclaration manuelle de routes supplémentaires. Tous les préfixes de routes seront générés automatiquement. Pour mettre à jour, changez simplement votre core.php.

```
//Forme ancienne:
Configure::write('Routing.admin', 'admin');

//à changer en:
Configure::write('Routing.prefixes', array('admin'));
```

Voir le guide des nouvelles fonctionnalités pour plus d'informations sur l'utilisation des préfixes de routes. Un petit changement a aussi été fait pour router les paramètres. Les paramètres routés doivent maintenant seulement être des caractères alphanumériques, - et _ ou `/[A-Z0-9-_+]+/`.

```
Router::connect('/:$%@#param/:action/*', array(...)); // BAD
Router::connect('/:can/:anybody/:see/:m-3/*', array(...)); //Acceptable
```

Dans 1.3, les entrailles du Router étaient hautement reconstruites pour améliorer la performance et réduire le fouillis du code. L'effet secondaire de cela est que deux fonctions rarement utilisées ont été supprimées, car ils étaient problématique et entraînait des bugs même avec le code de base existant. Les premiers segments de chemin utilisant les expressions régulières ont été retirés. Vous ne pouvez plus créer des routes comme :

```
Router::connect('/([0-9]+)-p-(.*)/', array('controller' => 'products', 'action' => 'show'))
```

Ces routes compliquent la compilation des routes et rendent impossibles les routes inversées. Si vous avez besoin de routes comme cela, il est recommandé que vous utilisiez les paramètres de route avec des patrons de capture. Le support de la next mid-route greedy star a été retirée. Il a été précedemment possible d'utiliser une greedy star dans le milieu de la route :

```
Router::connect(
    '/pages/*/:event',
    array('controller' => 'pages', 'action' => 'display'),
    array('event' => '[a-z0-9_-]+')
);
```

This is no longer supported as mid-route greedy stars behaved erratically, and complicated route compiling. Outside of these two edge-case features and the above changes the router behaves exactly as it did in 1.2.

Aussi, les personnes utilisant la clé 'id' dans les URLs en forme de tableau remarqueront que Router : :url() traite maintenant ceci en paramètre nommé. Si vous utilisiez précedemment cette approche pour passer le paramètre ID aux actions, vous aurez besoin de réécrire tous vos appels $html->link() et $this->redirect() pour refléter ce changement :

```
// format ancien:
$url = array('controller' => 'posts', 'action' => 'view', 'id' => $id);
// utilisations des cases:
Router::url($url);
$html->link($url);
$this->redirect($url);
// 1.2 result:
/posts/view/123
// 1.3 result:
/posts/view/id:123
// correct format:
$url = array('controller' => 'posts', 'action' => 'view', $id);
```

Dispatcher

`Dispatcher` n'est plus capable de définir un layout/viewPath de controller avec les paramètres de requête. Le Contrôle de ces propriétés devrait être géré par le Controller, pas le Dispatcher. Cette fonctionnalité n'était aussi pas documenté, et pas testé.

Debugger

- `Debugger::checkSessionKey()` a été renommé au profit de `Debugger::checkSecurityKeys()`
- Calling `Debugger::output("text")` ne fonctionne plus. Utilisez `Debugger::output("txt")`.

Object

- `Object::$_log` a été retiré. `CakeLog::write` est maintenant appelé statiquement. Regardez *Journalisation (logging)* pour plus d'informations sur les changements faits pour se connecter.

Sanitize

- `Sanitize::html()` retourne en général toujours des chaînes de caractère echappées. Dans le passé, utiliser le paramètre `$remove` would skip entity encoding, en retournant possiblement le contenu dangereux.
- `Sanitize::clean()` a maintenant une option `remove_html`. Cela déclenchera la fonctionnalité `strip_tags` de `Sanitize::html()`, et doit être utilisé en conjonction avec le paramètre `encode`.

Configure and App

- Configure : :listObjects() remplacé par App : :objects()
- Configure : :corePaths() remplacé par App : :core()
- Configure : :buildPaths() remplacé par App : :build()

- Configure ne gère plus les chemins.
- Configure : :write('modelPaths', array...) remplacé par App : :build(array('models' => array...))
- Configure : :read('modelPaths') remplacé par App : :path('models')
- Il n'y a plus de debug = 3. Le controller dumps generated by this setting often caused memory consumption issues making it an impractical and unusable setting. The `$cakeDebug` variable has also been removed from `View::renderLayout` You should remove this variable reference to avoid errors.
- `Configure::load()` can now load configuration files from plugins. Use `Configure::load('plugin.file');` to load configuration files from plugins. Any configuration files in your application that use `.` in the name should be updated to use _

Cache

In addition to being able to load CacheEngines from app/libs or plugins, Cache underwent some refactoring for CakePHP1.3. These refactorings focused around reducing the number and frequency of method calls. The end result was a significant performance improvement with only a few minor API changes which are detailed below.

The changes in Cache removed the singletons used for each Engine type, and instead an engine instance is made for each unique key created with `Cache::config()`. Since engines are not singletons anymore, `Cache::engine()` was not needed and was removed. In addition `Cache::isInitialized()` now checks cache *configuration names*, not cache *engine names*. You can still use `Cache::set()` or `Cache::engine()` to modify cache configurations. Also checkout the *Nouvelles caractéristiques dans CakePHP 1.3* for more information on the additional methods added to `Cache`.

It should be noted that using an app/libs or plugin cache engine for the default cache config can cause performance issues as the import that loads these classes will always be uncached. It is recommended that you either use one of the core cache engines for your `default` configuration, or manually include the cache engine class before configuring it. Furthermore any non-core cache engine configurations should be done in `app/config/bootstrap.php` for the same reasons detailed above.

Model Databases and Datasources

Model

- `Model::del()` and `Model::remove()` have been removed in favor of `Model::delete()`, which is now the canonical delete method.
- `Model::findAll`, findCount, findNeighbours, removed.
- Dynamic calling of setTablePrefix() has been removed. tableprefix should be with the `$tablePrefix` property, and any other custom construction behavior should be done in an overridden `Model::__construct()`.
- `DboSource::query()` now throws warnings for un-handled model methods, instead of trying to run them as queries. This means, people starting transactions improperly via the `$this->Model->begin()` syntax will need to update their code so that it accesses the model's DataSource object directly.
- Missing validation methods will now trigger errors in development mode.
- Missing behaviors will now trigger a cakeError.
- `Model::find(first)` will no longer use the id property for default conditions if no conditions are supplied and id is not empty. Instead no conditions will be used
- For Model : :saveAll() the default value for option 'validate' is now 'first' instead of true

Datasources

– DataSource : :exists() has been refactored to be more consistent with non-database backed datasources. Previously, if you set `var $useTable = false; var $useDbConfig = 'custom';`, it was impossible for `Model::exists()` to return anything but false. This prevented custom datasources from using `create()` or `update()` correctly without some ugly hacks. If you have custom datasources that implement `create()`, `update()`, and `read()` (since `Model::exists()` will make a call to `Model::find('count')`, which is passed to `DataSource::read()`), make sure to re-run your unit tests on 1.3.

Databases

Most database configurations no longer support the 'connect' key (which has been deprecated since pre-1.2). Instead, set `'persistent' => true` or false to determine whether or not a persistent database connection should be used

SQL log dumping

A commonly asked question is how can one disable or remove the SQL log dump at the bottom of the page ?. In previous versions the HTML SQL log generation was buried inside DboSource. For 1.3 there is a new core element called `sql_dump`. `DboSource` no longer automatically outputs SQL logs. If you want to output SQL logs in 1.3, do the following :

```
<?php echo $this->element('sql_dump'); ?>
```

You can place this element anywhere in your layout or view. The `sql_dump` element will only generate output when `Configure::read('debug')` is equal to 2. You can of course customize or override this element in your app by creating `app/views/elements/sql_dump.ctp`.

View and Helpers

View

– `View::renderElement` removed. Use `View::element()` instead.
– Automagic support for `.thtml` view file extension has been removed either declare `$this->ext = 'thtml';` in your controllers, or rename your views to use `.ctp`
– `View::set('title', $var)` no longer sets `$title_for_layout` when rendering the layout. `$title_for_layout` is still populated by default. But if you want to customize it, use `$this->set('title_for_layout', $var)`.
– `View::$pageTitle` has been removed. Use `$this->set('title_for_layout', $var);` instead.
– The `$cakeDebug` layout variable associated with debug = 3 has been removed. Remove it from your layouts as it will cause errors. Also see the notes related to SQL log dumping and Configure for more information.

All core helpers no longer use `Helper::output()`. The method was inconsistently used and caused output issues with many of FormHelper's methods. If you previously overrode `AppHelper::output()` to force helpers to auto-echo you will need to update your view files to manually echo helper output.

TextHelper

– `TextHelper::trim()` is deprecated, used `truncate()` instead.
– `TextHelper::highlight()` no longer has :
– an `$highlighter` parameter. Use `$options['format']` instead.
– an `$considerHtml``parameter. Use ``$options['html']` instead.

– `TextHelper::truncate()` no longer has :
– an `$ending` parameter. Use `$options['ending']` instead.
– an `$exact` parameter. Use `$options['exact']` instead.
– an `$considerHtml``parameter. Use ``$options['html']` instead.

PaginatorHelper

PaginatorHelper has had a number of enhancements applied to make styling easier. `prev()`, `next()`, `first()` and `last()`

The disabled state of these methods now defaults to `<span>` tags instead of `<div>` tags.

passedArgs are now auto merged with url options in paginator.

`sort()`, `prev()`, `next()` now add additional class names to the generated html. `prev()` adds a class of prev. `next()` adds a class of next. `sort()` will add the direction currently being sorted, either asc or desc.

FormHelper

– `FormHelper::dateTime()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
– `FormHelper::year()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
– `FormHelper::month()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
– `FormHelper::day()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
– `FormHelper::minute()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
– `FormHelper::meridian()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
– `FormHelper::select()` no longer has a `$showEmpty` parameter. Use `$attributes['empty']` instead.
– Default urls generated by form helper no longer contain 'id' parameter. This makes default urls more consistent with documented userland routes. Also enables reverse routing to work in a more intuitive fashion with default FormHelper urls.
– `FormHelper::submit()` Can now create other types of inputs other than type=submit. Use the type option to control the type of input generated.
– `FormHelper::button()` Now creates `<button>` elements instead of reset or clear inputs. If you want to generate those types of inputs use `FormHelper::submit()` with a `'type' => 'reset'` option for example.
– `FormHelper::secure()` and `FormHelper::create()` no longer create hidden fieldset elements. Instead they create hidden div elements. This improves validation with HTML4.

Also be sure to check the *Mises à jour 2.0* for additional changes and new features in the FormHelper.

HtmlHelper

– `HtmlHelper::meta()` no longer has an `$inline` parameter. It has been merged with the `$options` array.
– `HtmlHelper::link()` no longer has an `$escapeTitle` parameter. Use `$options['escape']` instead.
– `HtmlHelper::para()` no longer has an `$escape` parameter. Use `$options['escape']` instead.

– `HtmlHelper::div()` no longer has an `$escape` parameter. Use `$options['escape']` instead.
– `HtmlHelper::tag()` no longer has an `$escape` parameter. Use `$options['escape']` instead.
– `HtmlHelper::css()` no longer has an `$inline` parameter. Use `$options['inline']` instead.

SessionHelper

– `flash()` no longer auto echos. You must add an `echo $session->flash();` to your session->flash() calls. flash() was the only helper method that auto outputted, and was changed to create consistency in helper methods.

CacheHelper

CacheHelper's interactions with `Controller::$cacheAction` has changed slightly. In the past if you used an array for `$cacheAction` you were required to use the routed url as the keys, this caused caching to break whenever routes were changed. You also could set different cache durations for different passed argument values, but not different named parameters or query string parameters. Both of these limitations/inconsistencies have been removed. You now use the controller's action names as the keys for `$cacheAction`. This makes configuring `$cacheAction` easier as it's no longer coupled to the routing, and allows cacheAction to work with all custom routing. If you need to have custom cache durations for specific argument sets you will need to detect and update cacheAction in your controller.

TimeHelper

TimeHelper has been refactored to make it more i18n friendly. Internally almost all calls to date() have been replaced by strftime(). The new method TimeHelper : :i18nFormat() has been added and will take localization data from a LC_TIME locale definition file in app/locale following the POSIX standard. These are the changes made in the TimeHelper API :

– TimeHelper : :format() can now take a time string as first parameter and a format string as the second one, the format must be using the strftime() style. When called with this parameter order it will try to automatically convert the date format into the preferred one for the current locale. It will also take parameters as in 1.2.x version to be backwards compatible, but in this case format string must be compatible with date().
– TimeHelper : :i18nFormat() has been added

Deprecated Helpers

Both the JavascriptHelper and the AjaxHelper are deprecated, and the JsHelper + HtmlHelper should be used in their place.

You should replace

– `$javascript->link()` with `$html->script()`
– `$javascript->codeBlock()` with `$html->scriptBlock()` or `$html->scriptStart()` and `$html->scriptEnd()` depending on your usage.

Console and shells

Shell

`Shell::getAdmin()` has been moved up to `ProjectTask::getAdmin()`

Schema shell

– `cake schema run create` has been renamed to `cake schema create`
– `cake schema run update` has been renamed to `cake schema update`

Console Error Handling

The shell dispatcher has been modified to exit with a `1` status code if the method called on the shell explicitly returns `false`. Returning anything else results in a `0` status code. Before the value returned from the method was used directly as the status code for exiting the shell.

Shell methods which are returning `1` to indicate an error should be updated to return `false` instead.

`Shell::error()` has been modified to exit with status code 1 after printing the error message which now uses a slightly different formatting.

```
$this->error('Invalid Foo', 'Please provide bar.');
// outputs:
Error: Invalid Foo
Please provide bar.
// exits with status code 1
```

`ShellDispatcher::stderr()` has been modified to not prepend Error : to the message anymore. It's signature is now similar to `Shell::stdout()`.

ShellDispatcher : :shiftArgs()

The method has been modified to return the shifted argument. Before if no arguments were available the method was returning false, it now returns null. Before if arguments were available the method was returning true, it now returns the shifted argument instead.

Vendors, Test Suite & schema

vendors/css, vendors/js, and vendors/img

Support for these three directories, both in `app/vendors` as well as `plugin/vendors` has been removed. They have been replaced with plugin and theme webroot directories.

Test Suite and Unit Tests

Group tests should now extend TestSuite instead of the deprecated GroupTest class. If your Group tests do not run, you will need to update the base class.

Vendor, plugin and theme assets

Vendor asset serving has been removed in 1.3 in favour of plugin and theme webroot directories.

Schema files used with the SchemaShell have been moved to `app/config/schema` instead of `app/config/sql` Although `config/sql` will continue to work in 1.3, it will not in future versions, it is recommend that the new path is used.

Nouvelles caractéristiques dans CakePHP 1.3

CakePHP 1.3 introduit un nombre de nouvelles fonctionnalités. Ce guide tente de résumer ces changements et de pointer vers la documentation nouvelle quand c'est nécéssaire.

Components

SecurityComponent

Les différentes méthodes requireXX comme `requireGet` et `requirePost` acceptent maintenant un tableau unique en argument ainsi qu'une collection de noms en chaînes de caractère.

```
$this->Security->requirePost(array('edit', 'update'));
```

Paramètres du Component

Les paramètres du Component pour tous les components du coeur peuvent maintenant être définis à partir du tableau `$components`. Un peu comme les behaviors, vous pouvez déclarer les paramètres pour les components quand vous déclarer le component.

```
var $components = array(
    'Cookie' => array(
        'name' => 'MyCookie'
    ),
    'Auth' => array(
        'userModel' => 'MyUser',
        'loginAction' => array('controller' => 'users', 'action' => 'login')
    )
);
```

Ceci devrait réduire le désordre dans vos méthodes `beforeFilter()` de Controller.

EmailComponent

- Vous pouvez maintenant récupérer les contenus rendus des messages Email envoyés, en lisant `$this->Email->htmlMessage` et `$this->Email->textMessage`. Ces propriétés contiendront le contenu de l'email rendu correspondant à son nom.
- Many of EmailComponent's private methods have been made protected for easier extension.
- EmailComponent : :$to can now be an array. Allowing easier setting of multiple recipients, and consistency with other properties.
- `EmailComponent::$messageId` has been added, it allows control over the Message-ID header for email messages.

View & Helpers

Les Helpers peuvent maintenant être traités par `$this->Helper->func()` en plus de `$helper->func()`. Cela permet aux variables de vue et aux helpers de partager les noms et de ne pas créer de collisions.

Le nouveau JsHelper et les nouvelles fonctionnalités dans HtmlHelper

Regardez la *documentation de JsHelper* pour plus d'informations.

Pagination Helper

Le helper Pagination fournit des classes css supplémentaires pour le style et vous pouvez configurer la direction de sort() par défaut. `PaginatorHelper::next()` et `PaginatorHelper::prev()` générent maintenant des tags span par défaut, au lieu de divs.

Helper

`Helper::assetTimestamp()` a été ajoutée. Elle ajoutera des timestamps à tout asset sous WWW_ROOT. Elle fonctionne avec `Configure::read('Asset.timestamp');` comme avant, mais la fonctionnalité utilisée dans les helpers Html et Javascript a été rendué disponible pour tous les helpers. En supposant que `Asset.timestamp == force`

```
$path = 'css/cake.generic.css'
$stamped = $this->Html->assetTimestamp($path);

//$stamped contient 'css/cake.generic.css?5632934892'
```

Le timestamp ajouté contient la dernière modification de temps du fichier. Depuis que cette méthode est définie dans `Helper`, elle est disponible à toutes les sous-classes.

TextHelper

highlight() accepte maintenant un tableau de mots à surligner.

NumberHelper

Une nouvelle méthode `addFormat()` a été ajoutée. Cette méthode vous permet de configurer des ensembles de paramètres de monnaie, pour que vous n'ayez pas à les retaper.

```
$this->Number->addFormat('NOK', array('before' => 'Kr. '));
$formatted = $this->Number->currency(1000, 'NOK');
```

FormHelper

Le helper form a eu un certain nombre d'améliorations et de modifications de l'API, regardez les améliorations du Hemper Form (http ://book.cakephp.org/view/1616/x1-3-improvements) pour plus d'informations.

Logging

La connexion et `CakeLog` ont été améliorés considérablement, les deux dans les fonctionnalités et la flexibilité. Regardez New Logging features (http ://book.cakephp.org/view/1194/Logging) pour plus d'informations.

Caching

Les moteurs de Cache ont été fabriqués plus flexibles dans 1.3. Vous pouvez maintenant fournir des adapters de `Cache` personnalisés dans `app/libs` ainsi que dans les plugins en utilisant `$plugin/libs`. Les moteurs de cache App/plugin peuvent aussi surcharger les moteurs du coeur. Les adapters de Cache doivent être dans un répertoire de cache. Si vous aviez un moteur de cache nommé `MyCustomCacheEngine`, cela serait placé soit dans `app/libs/cache/my_custom_cache.php`, soit dans app/libs. Ou dans `$plugin/libs/cache/my_custom_cache.php` appartenant à un plugin. Les configs de Cache à partir des plugins ont besoin d'utiliser la syntaxe avec des points des plugins.

```
Cache::config('custom', array(
    'engine' => 'CachePack.MyCustomCache',
    ...
));
```

Les moteurs de cahce de App et Plugin doivent être configurés dans `app/bootstrap.php`. Si vous essayez de les configurer dans core.php, ils ne fonctionneront pas correctement.

Nouvelles méthodes de Cache

Cache a quelques nouvelles méthodes pour 1.3 ce qui rend l'introspection et le test bien plus facile.
- `Cache::configured()` retourne un tableau des clés de moteur de Cache configurés.
- `Cache::drop($config)` retire un moteur de Cache configuré. Une fois supprimé, les moteurs de cache ne sont plus lisible, et l'écriture n'est plus disponible.
- `Cache::increment()` Perform an atomic increment on a numeric value. This is not implemented in FileEngine.
- `Cache::decrement()` Perform an atomic decrement on a numeric value. This is not implemented in FileEngine.

Models, Behaviors and Datasource

App : :import(), datasources & datasources from plugins

Les sources de données peuvent maintenant être inclues chargées avec `App::import()` et être inclues dans les plugins ! Pour inclure un source de données dans votre plugin, vous pouvez la mettre dans `my_plugin/models/datasources/your_datasource.php`. Pour importer une Source de données à partir d'un plugin, utilisez `App::import('Datasource', 'MyPlugin.YourDatasource');`

Utiliser les sources de données dans votre database.php

Vous pouvez utiliser les sources de données de plugin en configurant la clé de la source de données avec le nom du plugin. Par exemple, si vous avez un plugin WebservicePack avec une source de données LastFm (plugin/webservice_pack/models/datasources/last_fm.php), vous pouvez faire :

```
var $lastFm = array(
    'datasource' => 'WebservicePack.LastFm'
    ...
```

Model
- Missing Validation methods now trigger errors, making debugging why validation isn't working easier.
- Models now support virtual fields (http ://book.cakephp.org/view/1608/Virtual-fields)

Behaviors

Using behaviors that do not exist, now triggers a `cakeError` making missing behaviors easier to find and fix.

CakeSchema

CakeSchema can now locate, read and write schema files to plugins. The SchemaShell also exposes this functionality, see below for changes to SchemaShell. CakeSchema also supports `tableParameters`.

Table Parameters are non column specific table information such as collation, charset, comments, and table engine type. Each Dbo implements the tableParameters they support.

tableParameters in MySQL

MySQL supports the greatest number of tableParameters ; You can use tableParameters to set a variety of MySQL specific settings.
– `engine` Control the storage engine used for your tables.
– `charset` Control the character set used for tables.
– `encoding` Control the encoding used for tables.
In addition to tableParameters MySQL dbo's implement `fieldParameters`. fieldParameters allow you to control MySQL specific settings per column.
– `charset` Set the character set used for a column
– `encoding` Set the encoding used for a column
See below for examples on how to use table and field parameters in your schema files.

tableParameters in Postgres

---

tableParameters in SQLite

Using tableParameters in schema files

You use `tableParameters` just as you would any other key in a schema file. Much like `indexes` :

```
var $comments => array(
    'id' => array('type' => 'integer', 'null' => false, 'default' => 0, 'key' => 'primary'
    'post_id' => array('type' => 'integer', 'null' => false, 'default' => 0),
    'comment' => array('type' => 'text'),
    'indexes' => array(
        'PRIMARY' => array('column' => 'id', 'unique' => true),
        'post_id' => array('column' => 'post_id'),
    ),
    'tableParameters' => array(
        'engine' => 'InnoDB',
        'charset' => 'latin1',
        'collate' => 'latin1_general_ci'
    )
);
```

is an example of a table using `tableParameters` to set some database specific settings. If you use a schema file that contains options and features your database does not implement, those options will be ignored. For example if you imported the above schema to a PostgreSQL server, all of the tableParameters would be ignore as PostgreSQL does not support any of the included options.

Console

Bake

Bake has had a number of significant changes made to it. Those changes are detailed in the bake updates section (http ://book.cakephp.org/view/1611/Bake-improvements-in-1-3)

Subclassing

The ShellDispatcher has been modified to not require shells and tasks to have *Shell* as their immediate parent anymore.

Output

`Shell::nl()` has been added. It returns a single or multiple linefeed sequences. `Shell::out()`, `err()` and `hr()` now accept a `$newlines` parameter which is passed to `nl()` and allows for controlling how newlines are appended to the output.

`Shell::out()` and `Shell::err()` have been modified, allowing a parameterless usage. This is especially useful if you're often using `$this->out("")` for outputting just a single newline.

Acl Shell

All AclShell commands now take `node` parameters. `node` parameters can be either an alias path like `controllers/Posts/view` or Model.foreign_key ie. `User.1`. You no longer need to know or use the aco/aro id for commands.

The Acl shell `dataSource` switch has been removed. Use the Configure settings instead.

SchemaShell

The Schema shell can now read and write Schema files and SQL dumps to plugins. It expects and will create schema files in `$plugin/config/schema`

Router and Dispatcher

Router

Generating urls with new style prefixes works exactly the same as admin routing did in 1.2. They use the same syntax and persist/behave in the same way. Assuming you have `Configure::write('Routing.prefixes', array('admin', 'member'));` in your core.php you will be able to do the following from a non-prefixed url :

```
$this->Html->link('Go', array('controller' => 'posts', 'action' => 'index', 'member' => tru
$this->Html->link('Go', array('controller' => 'posts', 'action' => 'index', 'admin' => true
```

Likewise, if you are in a prefixed url and want to go to a non-prefixed url, do the following :

```
$this->Html->link('Go', array('controller' => 'posts', 'action' => 'index', 'member' => fal
$this->Html->link('Go', array('controller' => 'posts', 'action' => 'index', 'admin' => fals
```

Route classes

For 1.3 the router has been internally rebuilt, and a new class `CakeRoute` has been created. This class handles the parsing and reverse matching of an individual connected route. Also new in 1.3 is the ability to create and use your own Route classes. You can implement any special routing features that may be needed in application routing classes. Developer route classes must extend `CakeRoute`, if they do not an error will be triggered. Commonly a custom route class will override the `parse()` and/or `match()` methods found in `CakeRoute` to provide custom handling.

Dispatcher

– Accessing filtered asset paths, while having no defined asset filter will create 404 status code responses.

Library classes

Inflector

You can now globally customize the default transliteration map used in Inflector : :slug using Inflector : :rules. eg. `Inflector::rules('transliteration', array('/å/' => 'aa', '/ø/' => 'oe'))`

The Inflector now also internally caches all data passed to it for inflection (except slug method).

Set

Set has a new method `Set::apply()`, which allows you to apply callbacks (http ://ca2.php.net/callback) to the results of `Set::extract` and do so in either a map or reduce fashion.

```
Set::apply('/Movie/rating', $data, 'array_sum');
```

Would return the sum of all Movie ratings in `$data`.

L10N

All languages in the catalog now have a direction key. This can be used to determine/define the text direction of the locale being used.

File

– File now has a copy() method. It copies the file represented by the file instance, to a new location.

Configure

– `Configure::load()` can now load configuration files from plugins. Use `Configure::load('plugin.file');` to load configuration files from plugins. Any configuration files in your application that use `.` in the name should be updated to used _

App/libs

In addition to `app/vendors` a new `app/libs` directory has been added. This directory can also be part of plugins, located at `$plugin/libs`. Libs directories are intended to contain 1st party libraries that do not come from 3rd parties or external vendors. This allows you to separate your organization's internal libraries from vendor libraries. `App::import()` has also been updated to import from libs directories.

```
App::import('Lib', 'ImageManipulation'); //imports app/libs/image_manipulation.php
```

You can also import libs files from plugins

```
App::import('Lib', 'Geocoding.Geocode'); //imports app/plugins/geocoding/libs/geocode.php
```

The remainder of lib importing syntax is identical to vendor files. So if you know how to import vendor files with unique names, you know how to import libs files with unique names.

Configuration

– The default `Security.level` in 1.3 is **medium** instead of **high**

– There is a new configuration value `Security.cipherSeed` this value should be customized to ensure more secure encrypted cookies, and a warning will be generated in development mode when the value matches its default value.

i18n

Now you can use locale definition files for the LC_TIME category to retrieve date and time preferences for a specific language. Just use any POSIX compliant locale definition file and store it at app/locale/*language*/ (do not create a folder for the category LC_TIME, just put the file in there).

For example, if you have access to a machine running debian or ubuntu you can find a french locale file at : /usr/share/i18n/locales/fr_FR. Copy the part corresponding to LC_TIME into app/locale/fr_fr/LC_TIME file. You can then access the time preferences for French language this way :

```
Configure::write('Config.language','fr-fr'); // set the current language
$monthNames = __c('mon',LC_TIME,true); // returns an array with the month names in French
$dateFormat = __c('d_fmt',LC_TIME,true); // return the preferred dates format for France
```

You can read a complete guide of possible values in LC_TIME definition file in this page (http ://sunsson.iptime.org/susv3/basedefs/xbd_chap07.html)

Miscellaneous

Error Handling

Subclasses of ErrorHandler can more easily implement additional error methods. In the past you would need to override `__construct()` and work around ErrorHandler's desire to convert all error methods into `error404` when debug = 0. In 1.3, error methods that are declared in subclasses are not converted to `error404`. If you want your error methods converted into error404, then you will need to do it manually.

Scaffolding

With the addition of `Routing.prefixes` scaffolding has been updated to allow the scaffolding of any one prefix.

```
Configure::write('Routing.prefixes', array('admin', 'member'));

class PostsController extends AppController {
    var $scaffold = 'member';
}
```

Would use scaffolding for member prefixed urls.

Validation

After 1.2 was released, there were numerous requests to add additional localizations to the `phone()` and `postal()` methods. Instead of trying to add every locale to Validation itself, which would result in large bloated ugly methods, and still not afford the flexibility needed for all cases, an alternate path was taken. In 1.3, `phone()` and `postal()` will pass off any country prefix it does not know how to handle to another class with the appropriate name. For example if you lived in the Netherlands you would create a class like

```
class NlValidation {
    public function phone($check) {
        ...
    }
    public function postal($check) {
        ...
    }
}
```

This file could be placed anywhere in your application, but must be imported before attempting to use it. In your model validation you could use your NlValidation class by doing the following.

```
public $validate = array(
    'phone_no' => array('rule' => array('phone', null, 'nl')),
    'postal_code' => array('rule' => array('postal', null, 'nl'))
);
```

When your model data is validated, Validation will see that it cannot handle the 'nl' locale and will attempt to delegate out to `NlValidation::postal()` and the return of that method will be used as the pass/fail for the validation. This approach allows you to create classes that handle a subset or group of locales, something that a large switch would not have. The usage of the individual validation methods has not changed, the ability to pass off to another validator has been added.

IP Address Validation

La validation des adresses IP a été étendu pour autoriser une stricte validation d'une Version d'IP spécifique. Cela utilisera aussi les méchanismes de validation natifs de PHP si ils sont disponibles.

```
Validation::ip($someAddress);          // Valide les deux IPv4 et IPv6
Validation::ip($someAddress, 'IPv4'); // Valide les adresses IPv4 seulement
Validation::ip($someAddress, 'IPv6'); // Valide les adresses IPv6 seulement
```

Validation : :uuid()

Un pattern de validation uuid() a été ajouté à la classe `Validation`. Cla vérifiera qu'une chaîne donnée correspondra à un uuid par pattern uniquement. Cela ne garantit pas l'unicité du uuid donné.

Informations générales

Le processus de développement CakePHP

Ici, nous tenterons d'expliquer le processus que nous utilisons lors de l'élaboration du CakePHP. Nous comptons beaucoup sur l'interaction communautaire par le biais billets et par le chat IRC. IRC est le meilleur endroit pour trouver des membres de l'équipe de développement et pour discuter d'idées, du dernier code, et de faire des commentaires généraux. Si quelque chose de plus formel doit être proposé ou s'il y a un problème avec une version sortie, le système de ticket est le meilleur endroit pour partager vos idées.

Nous maintenons 4 versions de CakePHP.

- **stable** : Versions taggées pour la production où la stabilité est plus importante que les fonctionnalités. Les questions déposées pour ces versions seront réglées dans la branche connexe, et feront parties de la prochaine version.
- **Branches de maintenance** : Les branches de Développement deviennent des branches de maintenance une fois qu'un niveau stable de la version a été atteint. Les branches de maintenance sont les endroits où toutes les corrections de bugs sont committées avant de faire leur chemin vers une version stable. Les branches de maintenance ont le même nom que la version principale pour lesquelles elles sont faites. Par ex : *1.2*. Si vous utilisez une version stable et avez besoin de correctifs qui n'ont pas fait leur chemin vers une version stable, vérifiez ici.
- **Branches de développement** : Les branches de Développement contiennent des correctifs de pointe et des fonctionnalités. Ils sont nommés d'après le numéro de version pour lesquels ils sont faits. Par ex : *1.3*. Une fois que les branches de développement ont atteint un niveau de version stable, elles deviennent des branches de maintenance, et plus aucune fonctionnalité nouvelle n'est introduite, à moins que ce soit absolument nécéssaire.
- **Branches de fonctionnalité** : Les branches de fonctionnalité contiennent des fonctionnalités non-finies, possiblement instable et sont recommandées uniquement pour les utilisateurs avertis interessés dans la fonctionnalité la plus avancée et qui souhaitent contribuer à la communauté. Les branches de fonctionnalité sont nommées selon la convention suivante de *version-fonctionnalité*. Un exemple serait *1.3-routeur* qui contiendrait de nouvelles fonctionnalités pour le Routeur dans 1.3.

Espèrons que cela vous aide à comprendre quel version est bonne pour vous. Une fois que vous choisissez votre version, vous vous sentirez peut-être contraints de contribuer à un report de bug ou de faire des commentaires généraux sur le code.

- Si vous utilisez une version stable ou une branche de maintenance, merci de soumettre des tickets ou discuter avec nous sur IRC.
- Si vous utilisez la branche de développement ou la branche de fonctionnalité, le premier endroit où aller est IRC. Si vous avez un commentaire et ne pouvez pas nous atteindre sur IRC après un jour ou deux, merci de nnous soumettre un ticket.

Si vous trouvez un problème, la meilleure réponse es d'écrire un test. Le meilleur conseil que nous pouvons offrir dans l'écriture des tests est de regarder ceux qui sont inclus dans le coeur.

Comme toujours, si vous avez n'importe quelle question ou commentaire, visitez nous sur #cakephp sur irc.freenode.net.

Glossaire

tableau routing Un tableau d'attributs qui sont passés au `Router::url()`. Typiquement, ils ressemblent à ça :

```
array('controller' => 'posts', 'action' => 'view', 5)
```

attributs html Un tableau de clé => valeurs qui sont composées dans les attributs html. Par exemple :

```
// Par exemple
array('class' => 'ma-classe', '_target' => 'blank')

// générerait
class="ma-classe" _target="blank"
```

Si une option peut être minimisée ou a le même nom que sa valeur, alors `true` peut être utilisée :

```
// Par exemple
array('checked' => true)

// Générerait
checked="checked"
```

syntaxe de plugin La syntaxe de Plugin fait référence au nom de la classe avec un point en séparation indiquant que les classes sont une partie d'un plugin. Par ex : `DebugKit.Toolbar`, le plugin est DebugKit, et le nom de classe est Toolbar.

notation avec points La notation avec points définit un chemin de tableau, en séparant les niveaux imbriqués avec `.` Par exemple :

```
Asset.filtre.css
```

Pointerait vers la valeur suivante :

```
array(
    'Asset' => array(
        'filtre' => array(
            'css' => 'vous m avez eu'
        )
    )
)
```

CSRF Les Reqûetes de site croisées de Contrefaçon. Emêche les attaques de replay, les soumissions doubles et les requêtes contrefaîtes provenant d'autres domaines.

routes.php Un fichier dans APP/Config qui contient la configuration de routing. Ce fichier est inclus avant que chaque requête soit traitée. Il devrait connecter toutes les routes dont votre applications a besoin afin que les requêtes puissent être routées aux contrôleur + action corrects.

DRY Ne vous répétez pas vous-même. Est un principe de développement de logiciel qui a pour objectif de réduire les répétitions d'information de tout type. Dans CakePHP, DRY est utilisé pout vous permettre de coder des choses et de les ré-utiliser à travers votre application.

CHAPITRE 14

Indices et tables

- *genindex*
- *modindex*

Index

Symbols

() (méthode), 372–379
:action, 578
:controller, 578
:plugin, 578
$this->request, 59
$this->response, 65
__() (global function), 209
__c() (global function), 210
__construct() (méthode Component), 79, 240
__d() (global function), 210
__dc() (global function), 210
__dcn() (global function), 210
__dn() (global function), 211
__n() (global function), 211

A

acceptLanguage() (méthode CakeRequest), 65
accepts() (méthode CakeRequest), 64
accepts() (méthode RequestHandlerComponent), 306
AclBehavior (class), 248
AclComponent (class), 273
action (RssHelper property), 399
addArgument() (méthode ConsoleOptionParser), 542
addArguments() (méthode ConsoleOptionParser), 542
addCrumb() (méthode HtmlHelper), 368
addDetector() (méthode CakeRequest), 64
addFormat() (méthode CakeNumber), 428
addFormat() (méthode NumberHelper), 382
addInputType() (méthode RequestHandlerComponent), 308
addOption() (méthode ConsoleOptionParser), 543
addOptions() (méthode ConsoleOptionParser), 543
addPathElement() (méthode Folder), 442
addScript() (méthode View), 92
addSubcommand() (méthode ConsoleOptionParser), 544
admin routing, 583
afterDelete() (méthode ModelBehavior), 192, 236
afterFilter() (méthode Controller), 50
afterFind() (méthode ModelBehavior), 192, 236
afterLayout() (méthode Helper), 103, 245
afterRender() (méthode Helper), 103, 245
afterRenderFile() (méthode Helper), 103, 245
afterSave() (méthode ModelBehavior), 192, 236
afterScaffoldSave() (méthode Controller), 53
afterScaffoldSaveError() (méthode Controller), 53
ajaxLogin (AuthComponent property), 299
alert() (méthode CakeLog), 467
allow() (méthode AuthComponent), 300
allowedActions (SecurityComponent property), 319
allowedControllers (SecurityComponent property), 319
alphaNumeric() (méthode Validation), 172
am() (global function), 211
App (class), 214
APP (global constant), 212
APP_DIR (global constant), 212
append() (méthode File), 447
append() (méthode View), 92
appError, 607
APPLIBS (global constant), 212

application exceptions, 605
apply() (méthode Set), 485, 512
AppShell (class), 548
assign() (méthode View), 92
attributs html, 764
AuthComponent (class), 287
authenticate (AuthComponent property), 299
authError (AuthComponent property), 299
authorize (AuthComponent property), 299
autoLink() (méthode TextHelper), 402
autoLinkEmails() (méthode TextHelper), 402
autoLinkUrls() (méthode TextHelper), 402
autoParagraph() (méthode TextHelper), 403

B

BadRequestException, 602
base (CakeRequest property), 65
base (RssHelper property), 399
baseUrl() (méthode Router), 594
beforeDelete() (méthode ModelBehavior), 192, 236
beforeFilter() (méthode Controller), 50
beforeFind() (méthode ModelBehavior), 192, 236
beforeLayout() (méthode Helper), 103, 245
beforeRedirect() (méthode Component), 79, 241
beforeRender() (méthode Component), 79, 240
beforeRender() (méthode Controller), 50
beforeRender() (méthode Helper), 103, 245
beforeRenderFile() (méthode Helper), 103, 245
beforeSave() (méthode ModelBehavior), 192, 236
beforeScaffold() (méthode Controller), 53
beforeValidate() (méthode ModelBehavior), 192, 236
between() (méthode Validation), 172
bindTranslation() (méthode TranslateBehavior), 259
blackHole() (méthode SecurityComponent), 318
blackHoleCallback (SecurityComponent property), 318
blank() (méthode Validation), 173
Blocks (View property), 93
blocks() (méthode View), 92
body() (méthode CakeResponse), 71
boolean() (méthode Validation), 173
build() (méthode App), 216
build() (méthode Xml), 524
buildFromArray() (méthode ConsoleOptionParser), 545
button() (méthode FormHelper), 349

C

Cache (class), 418
CACHE (global constant), 212
cache() (méthode CakeResponse), 71
CacheEngine (class), 415
CacheHelper (class), 325
CAKE (global constant), 212
CAKE_CORE_INCLUDE_PATH (global constant), 212
CakeBaseException, 604
CakeEmail (class), 420
CakeException, 604
CakeLog (class), 465
CakeNumber (class), 427
CakeRequest (class), 63
CakeResponse (class), 70
CakeRoute (class), 595
CakeSession (class), 600
CakeTime (class), 433
camelize() (méthode Inflector), 456
cc() (méthode Validation), 173
cd() (méthode Folder), 442
channel() (méthode RssHelper), 399
charset() (méthode CakeResponse), 71
charset() (méthode HtmlHelper), 354
check() (méthode Configure), 571
check() (méthode CookieComponent), 304
check() (méthode Hash), 476
check() (méthode SessionComponent), 323
check() (méthode SessionHelper), 400
check() (méthode Set), 486
checkbox() (méthode FormHelper), 343
checkNotModified() (méthode CakeResponse), 71
childCount() (méthode TreeBehavior), 268
children() (méthode TreeBehavior), 267
chmod() (méthode Folder), 442
cipher() (méthode Security), 468
classicExtract() (méthode Set), 486
classify() (méthode Inflector), 456
clean() (méthode Sanitize), 439
cleanInsert() (méthode String), 515
cleanup() (méthode ModelBehavior), 191, 235
clear() (méthode Cache), 419
clear() (méthode CacheEngine), 416
clear() (méthode Shell), 548
clearGroup() (méthode Cache), 420
clearGroup() (méthode CacheEngine), 416

clientIp() (méthode CakeRequest), 64
close() (méthode File), 447
combine() (méthode Hash), 472
combine() (méthode Set), 490
comparison() (méthode Validation), 174
compile() (méthode CakeRoute), 595
Component (class), 79, 240
components (AuthComponent property), 299
components (Controller property), 59
compress() (méthode CakeResponse), 71
config() (global function), 211
config() (méthode Cache), 418
config() (méthode CakeLog), 465
config() (méthode Configure), 572
ConfigReaderInterface (interface), 575
configuration, 567
Configure (class), 570
configured() (méthode CakeLog), 466
configured() (méthode Configure), 572
ConfigureException, 575
connect() (méthode Router), 592
connectNamed() (méthode Router), 593
ConsoleOptionParser (class), 540
constructAuthenticate() (méthode AuthComponent), 300
constructAuthorize() (méthode AuthComponent), 300
constructClasses() (méthode Controller), 54
ContainableBehavior (class), 250
contains() (méthode Hash), 475
contains() (méthode Set), 494
Controller (class), 50
convert() (méthode CakeTime), 433
convert() (méthode TimeHelper), 407
convertSlash() (global function), 211
convertSpecifiers() (méthode CakeTime), 433
convertSpecifiers() (méthode TimeHelper), 407
CookieComponent (class), 302
copy() (méthode File), 447
copy() (méthode Folder), 442
core() (méthode App), 215
core.php, 567
CORE_PATH (global constant), 212
correctSlashFor() (méthode Folder), 443
countDim() (méthode Set), 494
counter() (méthode PaginatorHelper), 391
create() (méthode File), 447
create() (méthode Folder), 443
create() (méthode FormHelper), 328
createFile() (méthode Shell), 548
critical() (méthode CakeLog), 467
CSRF, 765
csrfCheck (SecurityComponent property), 320
csrfExpires (SecurityComponent property), 320
csrfUseOnce (SecurityComponent property), 320
CSS (global constant), 212
css() (méthode HtmlHelper), 354
CSS_URL (global constant), 212
currency() (méthode CakeNumber), 427
currency() (méthode NumberHelper), 381
current() (méthode PaginatorHelper), 390
custom() (méthode Validation), 174

D

data (CakeRequest property), 65
data (RssHelper property), 399
data() (méthode CakeRequest), 64
database.php, 565
database.php.default, 565
date() (méthode Validation), 174
dateTime() (méthode FormHelper), 349
datetime() (méthode Validation), 175
DAY (global constant), 213
day() (méthode FormHelper), 351
dayAsSql() (méthode CakeTime), 433
dayAsSql() (méthode TimeHelper), 407
daysAsSql() (méthode CakeTime), 434
daysAsSql() (méthode TimeHelper), 408
debug() (global function), 211, 611
debug() (méthode CakeLog), 467
Debugger (class), 611
decimal() (méthode Validation), 176
decrement() (méthode Cache), 419
decrement() (méthode CacheEngine), 416
defaultCurrency() (méthode CakeNumber), 428
defaultCurrency() (méthode NumberHelper), 382
defaultLevels() (méthode CakeLog), 466
defaultModel() (méthode PaginatorHelper), 395
defaultRouteClass() (méthode Router), 594
delete() (méthode Cache), 419
delete() (méthode CacheEngine), 415
delete() (méthode CakeSession), 601
delete() (méthode Configure), 571
delete() (méthode CookieComponent), 305
delete() (méthode File), 447
delete() (méthode Folder), 443

delete() (méthode HttpSocket), 450
delete() (méthode SessionComponent), 323
deny() (méthode AuthComponent), 300
description() (méthode ConsoleOptionParser), 541
destroy() (méthode CookieComponent), 305
destroy() (méthode SessionComponent), 324
diff() (méthode Hash), 481
diff() (méthode Set), 495
dimensions() (méthode Hash), 479
dirsize() (méthode Folder), 444
disable() (méthode CakeLog), 467
disableCache() (méthode CakeResponse), 71
disableCache() (méthode Controller), 54
dispatchShell() (méthode Shell), 548
div() (méthode HtmlHelper), 361
docType() (méthode HtmlHelper), 356
document() (méthode RssHelper), 399
domain() (méthode CakeRequest), 63
domId() (méthode Helper), 103, 245
download() (méthode CakeResponse), 71
drop() (méthode CakeLog), 466
drop() (méthode Configure), 572
DRY, 765
DS (global constant), 212
dump() (méthode ConfigReaderInterface), 575
dump() (méthode Configure), 573
dump() (méthode Debugger), 611

E

elem() (méthode RssHelper), 399
element() (méthode View), 92
elementCache (View property), 93
email() (méthode Validation), 176
emailPattern() (méthode CakeEmail), 426
emergency() (méthode CakeLog), 467
enable() (méthode CakeLog), 467
enabled() (méthode CakeLog), 467
end() (méthode FormHelper), 331
end() (méthode View), 92
enum() (méthode Set), 496
env() (global function), 211
epilog() (méthode ConsoleOptionParser), 541
equalTo() (méthode Validation), 176
err() (méthode Shell), 548
error() (méthode FormHelper), 351
error() (méthode SessionHelper), 400
error() (méthode Shell), 548
errors() (méthode Folder), 444
escape() (méthode Sanitize), 440
etag() (méthode CakeResponse), 71
ExceptionRenderer (class), 604
excerpt() (méthode Debugger), 613
excerpt() (méthode String), 518
excerpt() (méthode TextHelper), 405
executable() (méthode File), 447
exists() (méthode File), 448
expand() (méthode Hash), 477
expires() (méthode CakeResponse), 71
exportVar() (méthode Debugger), 613
ext() (méthode File), 448
extend() (méthode View), 93
extension() (méthode Validation), 176
extract() (méthode Hash), 470
extract() (méthode Set), 497

F

fetch() (méthode View), 92
field (RssHelper property), 399
File (class), 447
file extensions, 585
file() (méthode CakeResponse), 71
file() (méthode FormHelper), 347
fileExistsInPath() (global function), 211
fileSize() (méthode Validation), 176
filter() (méthode Hash), 476
filter() (méthode Set), 498
find() (méthode Folder), 444
findRecursive() (méthode Folder), 444
first() (méthode PaginatorHelper), 390
flash (AuthComponent property), 299
flash() (méthode AuthComponent), 300
flash() (méthode Controller), 53
flash() (méthode SessionHelper), 401
flatten() (méthode Hash), 477
flatten() (méthode Set), 498
Folder (class), 442
Folder (File property), 447
Folder() (méthode File), 448
ForbiddenException, 602
format() (méthode CakeNumber), 431
format() (méthode CakeTime), 434
format() (méthode Hash), 474
format() (méthode NumberHelper), 385
format() (méthode Set), 499
format() (méthode TimeHelper), 408
formatDelta() (méthode CakeNumber), 432

formatDelta() (méthode NumberHelper), 386
FormHelper (class), 328
fromReadableSize() (méthode CakeNumber), 430
fromReadableSize() (méthode NumberHelper), 384
fromString() (méthode CakeTime), 435
fromString() (méthode TimeHelper), 409
FULL_BASE_URL (global constant), 212

G

gc() (méthode Cache), 420
gc() (méthode CacheEngine), 416
generateAuthKey() (méthode Security), 468
generateTreeList() (méthode TreeBehavior), 268
get() (méthode Hash), 470
get() (méthode HttpSocket), 450
getAjaxVersion() (méthode RequestHandlerComponent), 308
getCrumbList() (méthode HtmlHelper), 368
getCrumbs() (méthode HtmlHelper), 368
getInstance() (méthode Security), 468
getOptionParser() (méthode Shell), 549
getParentNode() (méthode TreeBehavior), 269
getPath() (méthode TreeBehavior), 269
getType() (méthode Debugger), 613
getVar() (méthode View), 91
getVars() (méthode View), 91
gmt() (méthode CakeTime), 435
gmt() (méthode TimeHelper), 409
group() (méthode File), 448
groupConfigs() (méthode Cache), 420

H

h() (global function), 211
handle (File property), 447
Hash (class), 470
hash() (méthode Security), 469
hasMethod() (méthode Shell), 549
hasNext() (méthode PaginatorHelper), 391
hasPage() (méthode PaginatorHelper), 391
hasPrev() (méthode PaginatorHelper), 391
hasTask() (méthode Shell), 549
header() (méthode CakeRequest), 64
header() (méthode CakeResponse), 70
Helper (class), 102, 245
helpers (Controller property), 58
helpers (RssHelper property), 399
here (CakeRequest property), 65
here (RssHelper property), 399
hidden() (méthode FormHelper), 342
highlight() (méthode String), 515
highlight() (méthode TextHelper), 403
host() (méthode CakeRequest), 63
HOUR (global constant), 213
hour() (méthode FormHelper), 351
hr() (méthode Shell), 549
html() (méthode Sanitize), 440
HtmlHelper (class), 353
HttpSocket (class), 450
humanize() (méthode Inflector), 456

I

i18nFormat() (méthode CakeTime), 435
i18nFormat() (méthode TimeHelper), 409
identify() (méthode AuthComponent), 300
image() (méthode HtmlHelper), 357
IMAGES (global constant), 212
IMAGES_URL (global constant), 212
import() (méthode App), 218
in() (méthode Shell), 549
inactiveMins() (méthode Security), 469
inCakePath() (méthode Folder), 445
increment() (méthode Cache), 419
increment() (méthode CacheEngine), 416
Inflector (class), 456
info (File property), 447
info() (méthode CakeLog), 467
info() (méthode File), 448
IniReader (class), 576
init() (méthode App), 220
initialize() (méthode AuthComponent), 301
initialize() (méthode Component), 79, 240
initialize() (méthode Shell), 549
inList() (méthode Validation), 177
inPath() (méthode Folder), 445
input() (méthode CakeRequest), 64
input() (méthode FormHelper), 332
inputs() (méthode FormHelper), 334
insert() (méthode Hash), 471
insert() (méthode Set), 500
insert() (méthode String), 515
InternalErrorException, 602
invoke() (méthode Debugger), 613
ip() (méthode Validation), 177
is() (méthode CakeRequest), 64
isAbsolute() (méthode Folder), 445

isAtom() (méthode RequestHandlerComponent), 307
isAuthorized() (méthode AuthComponent), 301
isFieldError() (méthode FormHelper), 351
isFuture() (méthode CakeTime), 438
isFuture() (méthode TimeHelper), 412
isMobile() (méthode RequestHandlerComponent), 307
isPast() (méthode CakeTime), 439
isPast() (méthode TimeHelper), 412
isRss() (méthode RequestHandlerComponent), 307
isSlashTerm() (méthode Folder), 445
isThisMonth() (méthode CakeTime), 438
isThisMonth() (méthode TimeHelper), 412
isThisWeek() (méthode CakeTime), 438
isThisWeek() (méthode TimeHelper), 412
isThisYear() (méthode CakeTime), 438
isThisYear() (méthode TimeHelper), 412
isToday() (méthode CakeTime), 438
isToday() (méthode TimeHelper), 412
isTomorrow() (méthode CakeTime), 438
isTomorrow() (méthode TimeHelper), 412
isWap() (méthode RequestHandlerComponent), 308
isWindowsPath() (méthode Folder), 445
isXml() (méthode RequestHandlerComponent), 307
item() (méthode RssHelper), 400
items() (méthode RssHelper), 400

J

JS (global constant), 213
JS_URL (global constant), 213
JsonView (class), 98

L

label() (méthode FormHelper), 342
last() (méthode PaginatorHelper), 390
lastAccess() (méthode File), 448
lastChange() (méthode File), 448
layout (View property), 93
levels() (méthode CakeLog), 466
link() (méthode HtmlHelper), 358
link() (méthode PaginatorHelper), 394
listTimezones() (méthode CakeTime), 438
listTimezones() (méthode TimeHelper), 412
load() (méthode App), 220
load() (méthode Configure), 573
loadConfig() (méthode HtmlHelper), 368
loadModel() (méthode Controller), 57
loadTasks() (méthode Shell), 549
location() (méthode App), 215
location() (méthode CakeResponse), 70
lock (File property), 447
log() (méthode Debugger), 612
LOG_ERROR (global constant), 570
LogError() (global function), 211
loggedIn() (méthode AuthComponent), 301
login() (méthode AuthComponent), 301
loginAction (AuthComponent property), 300
loginRedirect (AuthComponent property), 300
logout() (méthode AuthComponent), 301
logoutRedirect (AuthComponent property), 300
LOGS (global constant), 213
luhn() (méthode Validation), 177

M

map() (méthode Hash), 480
map() (méthode Set), 501
mapActions() (méthode AuthComponent), 301
mapResources() (méthode Router), 594
match() (méthode CakeRoute), 595
matches() (méthode Set), 503
maxDimensions() (méthode Hash), 480
maxLength() (méthode Validation), 178
md5() (méthode File), 448
media() (méthode HtmlHelper), 360
MediaView (class), 95
merge() (méthode Hash), 478
merge() (méthode Set), 504
mergeDiff() (méthode Hash), 481
meridian() (méthode FormHelper), 351
messages() (méthode Folder), 445
meta() (méthode HtmlHelper), 355
method() (méthode CakeRequest), 63
MethodNotAllowedException, 602
mime() (méthode File), 449
mimeType() (méthode Validation), 178
minLength() (méthode Validation), 178
MINUTE (global constant), 213
minute() (méthode FormHelper), 351
MissingActionException, 603
MissingBehaviorException, 603
MissingComponentException, 603
MissingConnectionException, 603
MissingControllerException, 603
MissingDatabaseException, 603
MissingHelperException, 603

MissingLayoutException, 603
MissingShellException, 603
MissingShellMethodException, 603
MissingTableException, 603
MissingTaskException, 603
MissingViewException, 603
mode (Folder property), 442
model (RssHelper property), 399
ModelBehavior (class), 191, 235
modified() (méthode CakeResponse), 71
money() (méthode Validation), 178
MONTH (global constant), 213
month() (méthode FormHelper), 350
move() (méthode Folder), 445
moveDown() (méthode TreeBehavior), 269
moveUp() (méthode TreeBehavior), 270
multiple() (méthode Validation), 178

N

name (Controller property), 58
name (File property), 447
name() (méthode File), 448
named parameters, 587
naturalNumber() (méthode Validation), 179
nest() (méthode Hash), 483
nest() (méthode Set), 513
nestedList() (méthode HtmlHelper), 363
next() (méthode PaginatorHelper), 390
nice() (méthode CakeTime), 436
nice() (méthode TimeHelper), 409
niceShort() (méthode CakeTime), 436
niceShort() (méthode TimeHelper), 410
nl() (méthode Shell), 549
normalize() (méthode Hash), 482
normalize() (méthode Set), 505
normalizePath() (méthode Folder), 445
notation avec points, 765
notEmpty() (méthode Validation), 179
NotFoundException, 602
notice() (méthode CakeLog), 467
NotImplementedException, 602
NumberHelper (class), 381
numbers() (méthode PaginatorHelper), 388
numeric() (méthode Hash), 479
numeric() (méthode Set), 507
numeric() (méthode Validation), 179

O

objects() (méthode App), 217
offset() (méthode File), 448
onlyAllow() (méthode CakeRequest), 63
open() (méthode File), 448
options() (méthode PaginatorHelper), 392
out() (méthode Shell), 549
output (View property), 93
owner() (méthode File), 449

P

paginate() (méthode Controller), 55
PaginatorComponent (class), 311
PaginatorHelper (class), 387
para() (méthode HtmlHelper), 361
param() (méthode CakeResponse), 71
param() (méthode PaginatorHelper), 395
params (CakeRequest property), 65
params (RssHelper property), 399
params() (méthode PaginatorHelper), 395
paranoid() (méthode Sanitize), 441
parse() (méthode CakeRoute), 595
parseExtensions() (méthode Router), 594
passed arguments, 586
password() (méthode AuthComponent), 301
password() (méthode FormHelper), 342
patch() (méthode HttpSocket), 451
path (File property), 447
path (Folder property), 442
path() (méthode App), 215
paths() (méthode App), 215
perms() (méthode File), 449
phone() (méthode Validation), 179
PhpReader (class), 576
plugin routing, 584
pluginPath() (méthode App), 217
pluginSplit() (global function), 211
pluralize() (méthode Inflector), 456
post() (méthode HttpSocket), 450
postal() (méthode Validation), 180
postButton() (méthode FormHelper), 349
postConditions() (méthode Controller), 54
postLink() (méthode FormHelper), 349
pr() (global function), 211
precision() (méthode CakeNumber), 429
precision() (méthode NumberHelper), 383
prefers() (méthode RequestHandlerComponent), 310
prefix routing, 583

prepare() (méthode File), 449
prepend() (méthode View), 92
prev() (méthode PaginatorHelper), 389
PrivateActionException, 603
promote() (méthode Router), 594
pushDiff() (méthode Set), 508
put() (méthode HttpSocket), 450
pwd() (méthode File), 449
pwd() (méthode Folder), 446

Q

query (CakeRequest property), 65
query() (méthode CakeRequest), 64

R

radio() (méthode FormHelper), 344
range() (méthode Validation), 180
read() (méthode Cache), 418
read() (méthode CacheEngine), 415
read() (méthode CakeSession), 601
read() (méthode ConfigReaderInterface), 575
read() (méthode Configure), 571
read() (méthode CookieComponent), 304
read() (méthode File), 449
read() (méthode Folder), 446
read() (méthode SessionComponent), 323
read() (méthode SessionHelper), 400
readable() (méthode File), 449
realpath() (méthode Folder), 446
recover() (méthode TreeBehavior), 271
redirect() (méthode AuthComponent), 301
redirect() (méthode Controller), 52
redirect() (méthode Router), 593
redirectUrl() (méthode AuthComponent), 301
reduce() (méthode Hash), 480
referer() (méthode CakeRequest), 63
referer() (méthode Controller), 54
remove() (méthode Hash), 471
remove() (méthode Set), 508
removeFromTree() (méthode TreeBehavior), 270
render() (méthode Controller), 51
renderAs() (méthode RequestHandlerComponent), 310
reorder() (méthode TreeBehavior), 271, 272
request (AuthComponent property), 300
request (View property), 93
request() (méthode HttpSocket), 451
requestAction() (méthode Controller), 55
RequestHandlerComponent (class), 306
requireAuth() (méthode SecurityComponent), 319
requireDelete() (méthode SecurityComponent), 319
requireGet() (méthode SecurityComponent), 319
requirePost() (méthode SecurityComponent), 319
requirePut() (méthode SecurityComponent), 319
requireSecure() (méthode SecurityComponent), 319
reset() (méthode Inflector), 457
respondAs() (méthode RequestHandlerComponent), 310
response (AuthComponent property), 300
responseHeader() (méthode CakeBaseException), 604
responseType() (méthode RequestHandlerComponent), 310
restore() (méthode Configure), 574
reverse() (méthode Set), 509
RFC
    RFC 2616, 452, 725
    RFC 4122, 514
rijndael() (méthode Security), 468
ROOT (global constant), 213
Router (class), 592
routes.php, 578, 765
RssHelper (class), 396
rules() (méthode Inflector), 457
runCommand() (méthode Shell), 550

S

safe() (méthode File), 449
Sanitize (class), 439
scaffoldError() (méthode Controller), 53
script() (méthode HtmlHelper), 361
scriptBlock() (méthode HtmlHelper), 363
scriptEnd() (méthode HtmlHelper), 363
scriptStart() (méthode HtmlHelper), 363
SECOND (global constant), 213
secure() (méthode FormHelper), 352
Security (class), 468
SecurityComponent (class), 317
select() (méthode FormHelper), 344
send() (méthode CakeResponse), 71
serverOffset() (méthode CakeTime), 436
serverOffset() (méthode TimeHelper), 410
SessionComponent (class), 323
SessionHelper (class), 400
sessionKey (AuthComponent property), 300
Set (class), 485

set() (méthode Cache), 419
set() (méthode Controller), 50
set() (méthode View), 91
setContent() (méthode RequestHandlerComponent), 309
setExtensions() (méthode Router), 594
setFlash() (méthode SessionComponent), 324
setHash() (méthode Security), 469
setup() (méthode ModelBehavior), 191, 235
sharable() (méthode CakeResponse), 71
Shell (class), 548
shortPath() (méthode Shell), 550
shutdown() (méthode App), 220
shutdown() (méthode AuthComponent), 301
shutdown() (méthode Component), 79, 240
singularize() (méthode Inflector), 456
size() (méthode File), 449
slashTerm() (méthode Folder), 446
slug() (méthode Inflector), 457
sort (Folder property), 442
sort() (méthode Hash), 480
sort() (méthode PaginatorHelper), 387
sort() (méthode Set), 511
sortByKey() (global function), 212
sortDir() (méthode PaginatorHelper), 388
sortKey() (méthode PaginatorHelper), 388
ssn() (méthode Validation), 180
start() (méthode View), 92
startIfEmpty() (méthode View), 92
startup() (méthode AuthComponent), 301
startup() (méthode Component), 79, 240
startup() (méthode Shell), 550
statusCode() (méthode CakeResponse), 71
store() (méthode Configure), 574
stream() (méthode CakeLog), 467
String (class), 514
stripLinks() (méthode String), 516
stripLinks() (méthode TextHelper), 403
stripslashes_deep() (global function), 212
style() (méthode HtmlHelper), 357
subdomains() (méthode CakeRequest), 63
submit() (méthode FormHelper), 348
syntaxe de plugin, 765

T

tableau routing, 764
tableCells() (méthode HtmlHelper), 365
tableHeaders() (méthode HtmlHelper), 364
tableize() (méthode Inflector), 456
tag() (méthode HtmlHelper), 360
tagIsInvalid() (méthode FormHelper), 351
tail() (méthode String), 517
tail() (méthode TextHelper), 404
tasks (Shell property), 548
TESTS (global constant), 213
text() (méthode FormHelper), 342
textarea() (méthode FormHelper), 343
TextHelper (class), 402
themePath() (méthode App), 217
time() (méthode RssHelper), 400
time() (méthode Validation), 180
TIME_START (global constant), 213
timeAgoInWords() (méthode CakeTime), 436
timeAgoInWords() (méthode TimeHelper), 410
TimeHelper (class), 406
timezone() (méthode CakeTime), 438
timezone() (méthode TimeHelper), 412
TMP (global constant), 213
toArray() (méthode Xml), 524
toAtom() (méthode CakeTime), 437
toAtom() (méthode TimeHelper), 411
tokenize() (méthode String), 514
toList() (méthode String), 518
toList() (méthode TextHelper), 406
toPercentage() (méthode CakeNumber), 430
toPercentage() (méthode NumberHelper), 384
toQuarter() (méthode CakeTime), 437
toQuarter() (méthode TimeHelper), 411
toReadableSize() (méthode CakeNumber), 430
toReadableSize() (méthode NumberHelper), 384
toRSS() (méthode CakeTime), 438
toRSS() (méthode TimeHelper), 412
toServer() (méthode CakeTime), 438
toServer() (méthode TimeHelper), 412
toUnix() (méthode CakeTime), 438
toUnix() (méthode TimeHelper), 412
trace() (méthode Debugger), 612
TranslateBehavior (class), 256
tree() (méthode Folder), 446
TreeBehavior (class), 262, 267
truncate() (méthode String), 516
truncate() (méthode TextHelper), 404
type() (méthode CakeResponse), 71
type() (méthode CookieComponent), 305

U

unauthorizedRedirect (AuthComponent property), 300
underscore() (méthode Inflector), 456
unlockedFields (SecurityComponent property), 319
unlockField() (méthode FormHelper), 352
uploadError() (méthode Validation), 180
url() (méthode Helper), 102, 245
url() (méthode HtmlHelper), 366
url() (méthode PaginatorHelper), 394
url() (méthode Router), 594
url() (méthode Validation), 181
user() (méthode AuthComponent), 302
userDefined() (méthode Validation), 181
uses (Controller property), 58
uses (Shell property), 548
uses() (méthode App), 214
useTag() (méthode HtmlHelper), 367
uuid() (méthode String), 514
uuid() (méthode Validation), 181
uuid() (méthode View), 92

V

valid() (méthode SessionHelper), 401
validateAuthKey() (méthode Security), 469
validatePost (SecurityComponent property), 320
Validation (class), 172
ValidationisUnique() (méthode ModelValidation), 177
value() (méthode Helper), 102, 245
variable() (méthode Inflector), 457
VENDORS (global constant), 213
verify() (méthode TreeBehavior), 272
version (RssHelper property), 399
version() (méthode Configure), 571
View (class), 91
viewClassMap() (méthode RequestHandlerComponent), 311

W

wasWithinLast() (méthode CakeTime), 439
wasWithinLast() (méthode TimeHelper), 412
wasYesterday() (méthode CakeTime), 438
wasYesterday() (méthode TimeHelper), 412
webroot (CakeRequest property), 65
webroot() (méthode Helper), 102, 245
WEBROOT_DIR (global constant), 213
WEEK (global constant), 213
wrap() (méthode String), 515
wrapText() (méthode Shell), 550
writable() (méthode File), 449
write() (méthode Cache), 419
write() (méthode CacheEngine), 415
write() (méthode CakeLog), 466
write() (méthode CakeSession), 601
write() (méthode Configure), 570
write() (méthode CookieComponent), 303
write() (méthode File), 449
write() (méthode SessionComponent), 323
WWW_ROOT (global constant), 213

X

Xml (class), 519
XmlView (class), 98

Y

YEAR (global constant), 213
year() (méthode FormHelper), 350