Le Manuel :: Collection 1.3

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Welcome to the Cookbook

- The CakePHP 1.3 Book
- The CakePHP 1.2 Book
- The CakePHP 1.1 Book

- Clear Local Cache

Le Manuel

Bienvenue dans le *Cookbook* (Livre de cuisine), la documentation CakePHP. Le Cookbook est un système wiki, permettant les contributions publiques. Avec un système ouvert, nous espérons maintenir un haut niveau de qualité, de validité et de précision. Le Cookbook est aussi fait de telle sorte qu'il est facile à tout un chacun de contribuer.

Un ***énorme*** merci à AD7six qui s'est dévoué corps et âme au Cookbook, en y consacrant d'innombrables heures de développement, de test et d'amélioration.

Comment vous pouvez aider

Si vous notez une erreur, quelque chose qui est incomplet, quelque chose qui n'a pas encore été entièrement couvert ou quelque chose qui n'est tout simplement pas écrit comme vous le souhaiteriez, voici comment vous pouvez aider :

1. Cliquez sur le lien Editer de la section que vous souhaitez modifier.
2. Connectez-vous au Cookbook si nécessaire, avec votre compte Bakery. Tout le monde peut obtenir un compte Bakery !
3. Relisez s'il vous plaît les directives à respecter lors du dépôt de vos propositions au Cookbook, pour vous assurer de leur cohérence.
4. Soumettez vos ajouts/éditions en utilisant du HTML valide et sémantique.
5. Suivez la progression de vos soumissions en utilisant les flux rss ou en venant vérifier ici le jour suivant, pour voir si vos modifications sont approuvées.

Traductions

Envoyez un email à l'équipe documentation (docs at cakephp dot org) ou retrouvez-le sur IRC (#cakephp sur freenode) pour discutez de tout effort de traduction auquel vous aimeriez participer.

Note des traducteurs francophones : une section du forum du portail CakePHP-fr.org est dédié à la traduction du CookBook. Il regroupe des informations et des conseils pour participer à cet effort. Avant de vous lancer dans un effort de traduction, nous vous invitons à souscrire à la mailing-list des traducteurs francophones pour vous présenter et poser vos questions !

Astuces pour les traducteurs :

- Naviguez dans le manuel et éditez-le dans la langue pour laquelle vous voulez traduire le contenu, sinon vous ne verrez pas ce qui a déjà été traduit.
- N'hésitez pas à vous investir si votre langue existe déjà dans le manuel.
- Utilisez le menu *todo* (tout en haut à droite) pour voir quel partie réclame une attention particulière dans votre langue.
- Utilisez les Formes informelles.
- Traduisez en même temps le contenu et le titre d'une section.
- Traduisez également les liens des références croisées pour qu'elles pointent vers des sections traduites (ou à traduire).
- Modifiez les liens de références croisées pour qu'ils pointent vers les pages traduites.
- Comparez avec le contenu anglais avant de soumettre une correction (si vous corrigez quelque chose, mais que vous n'intégrez pas une 'remontée' de la modification, votre soumission ne sera pas acceptée).
- N'utilisez pas les entités HTML pour les caractères accentués, le manuel utilise l'UTF-8.
- Si vous avez besoin d'écrire un terme anglais, entourez-le avec des balises . Par exemple : "abcd abcd *Controller* abcd..." or "abcd abcd Contrôleur (*Controller*) abcd..." selon les besoins.
- Ne soumettez pas de traductions partielles
- N'éditez pas une rubrique pour laquelle un changement est en cours.
- Ne changez pas le balisage de manière significative et n'ajoutez pas de nouveaux contenus. S'il manque une info dans la version originale, soumettez d'abord une mise à jour pour celle-ci.

Nous nous sommes engagés à fournir la meilleure documentation existante pour CakePHP. Nous espérons que vous vous joindrez à nous en utilisant *le CookBook* et en nous faisant vos retours sur ce projet, dont nous avons tous beaucoup bénéficié.

1 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.

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

CakePHP est un framework de développement rapide pour PHP, gratuit et open-source. C'est un ensemble de briques élémentaires pour les programmeurs qui créent des applications web. Notre objectif principal est de vous permettre de travailler de manière rapide et structurée, sans toutefois perdre en 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 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
- Système de license souple
- Compatible avec les versions 4 et 5 de PHP
- Fonctions CRUD (create, read, update, delete) intégrées pour les interactions avec la base de données
- Scaffolding (maquettage rapide) d'application
- Génération de code
- Architecture MVC
- Dispatcheur de requêtes avec des URLs propres et personnalisables grâce un système de routes
- Validation intégrée des données
- Système de template 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 flexible
- Nettoyage des données
- Système de cache souple
- Localisation et internationalisation
- Fonctionne sur n'importe quelle arborescence de site web, avec un zest de configuration Apache pas très compliquée

1.2 Où trouver de l'aide ?

http://www.cakephp.org

Le site officiel 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 être probablement le premier endroit où vous rendre pour obtenir des réponses. Comme pour beaucoup d'autres projets open source, nous accueillons de nouvelles personnes régulièrement. Faites 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 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 familiarisé avec CakePHP, connectez-vous pour partager vos connaissances avec la communauté et obtenez en un instant la gloire et la fortune.

L'API

http://api.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.

La forge Cake

http://www.cakeforge.org

La forge Cake est une autre ressource développeur que vous pouvez utiliser pour héberger vos projets CakePHP et les partager avec d'autres. Si vous recherchez (ou voulez partager) un composant de tueur ou un plugin digne d'éloges, rendez-vous sur CakeForge.

Les cas de Test

http://api.cakephp.org/tests

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 1.2. Ils peuvent servir d'exemples pratiques pour l'utilisation d'une fonction et de donnée membres d'une classe. Pour obtenir les cas de test du core, vous devez télécharger un paquet du jour (*nightly package*) ou réaliser un *checkout* d'une branche svn. Les cas de test seront situés sous :

```
cake/tests/cases
```

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 s'y trouve habituellement, en particulier durant les heures du jour pour les utilisateurs d'Amérique du Nord et du Sud. Nous serions ravi 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 voiture sportive de marque.

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.

1.3 Comprendre le modèle M-V-C

CakePHP suit le motif de conception logicielle MVC. Programmer en utilisant MVC sépare votre application en 3 parties principales :

1. Le Modèle représente les données de l'application
2. La Vue affiche une présentation des données du modèle
3. Le Contrôleur intercepte et route les requêtes faites par le client

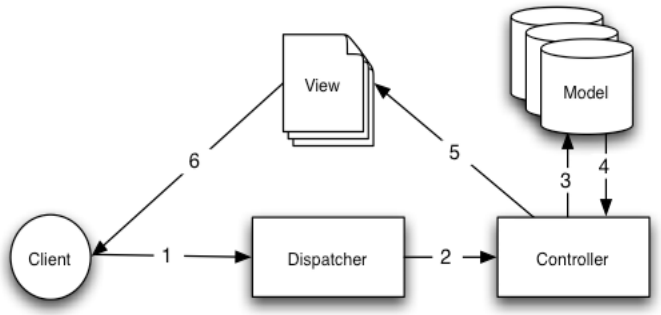

Figure: 1: Une requête MVC basique

La figure 1 montre un exemple de requête MVC sommaire avec CakePHP. Pour illustrer cela, supposons qu'un client nommé "Ricardo" a simplement cliqué le lien "Achetez un Cake personnalisé maintenant !" sur la page d'accueil de votre application.

- Ricardo clique le lien qui pointe vers *http://www.exemple.com/cakes/acheter* et son navigateur envoie une requête à votre serveur web.
- Le répartiteur (*dispatcher*) analyse l'URL (/cakes/acheter) et transmet la requête au contrôleur concerné
- Le contrôleur exécute la logique spécifique de l'application. Par exemple, il peut vérifier si Ricardo s'est identifié.

- Le contrôleur utilise aussi les modèles pour obtenir l'accès aux données de l'application. Le plus souvent, les modèles représentent une table de la base de données, mais ils peuvent aussi représenter des entrées LDAP, des flux RSS ou des fichiers sur l'ordinateur. Dans cet exemple, le contrôleur utilise un modèle qui récupère, dans la base de données, les derniers achats de Ricardo.
- Une fois que le contrôleur a effectué ses traitements « magiques » sur les données, il les transmet à une vue. La vue récupère ces données et les formate pour les présenter à l'utilisateur. Les vues dans CakePHP sont le plus souvent au format HTML, mais une vue pourrait tout aussi bien être, en fonction de vos besoins, un document PDF ou XML, ou bien un objet JSON.
- Une fois que la vue a utilisé les données du contrôleur pour construire une vue de rendu complète, le contenu de cette vue est renvoyé au navigateur de Ricardo.

Presque chaque requête à votre application suivra ce schéma de base. Nous le retrouverons plus tard, dans de nombreux cas particuliers de Cake, gardez-le donc dans un coin de votre esprit tandis que nous poursuivons.

Avantages

Pourquoi utiliser MVC ?

Parce que c'est un vrai motif de conception logiciel éprouvé, qui transforme une application en un ensemble maintenable, modulaire et rapidement développé. Façonner les tâches applicatives dans des modèles, vues et contrôleurs séparés, permet à votre application de se sentir très "à l'aise dans ses baskets". Les nouvelles fonctionnalités sont ajoutées facilement et modifier les anciennes est un jeu d'enfant ! L'architecture modulaire et séparée offre également aux développeurs et designers, la possibilité de travailler en parallèle, avec la capacité de créer rapidement un prototype. La séparation permet aussi aux développeurs de faire des modifications sur une partie de l'application sans affecter les autres.

Si vous n'avez jamais développé une application de cette manière, cela demande un peu de pratique, mais nous sommes certains qu'une fois votre première application construite avec CakePHP, vous ne voudrez jamais revenir en arrière.

2 Principes de base de CakePHP

Le framework CakePHP fournit une base robuste pour votre application. Il peut se charger de chaque aspect, depuis la requête initiale de l'utilisateur jusqu'au rendu final d'une page web. Et puisque le framework suit les principes du motif MVC, il vous permet de personnaliser et d'étendre facilement la plupart des aspects de votre application.

Le framework fournit également une structure organisationnelle basique, des noms de fichiers aux noms de table d'une base de données, assurant consistance et logique à votre application tout entière. Ce concept est simple mais puissant. Suivez les conventions et vous saurez toujours exactement où se trouvent les choses et comment elles sont organisées.

2.1 Structure de CakePHP

Un Composant (*Component*) est une classe qui s'intègre dans la logique du contrôleur. Si vos contrôleurs ou vos applications doivent partager une logique, alors créer un Composant 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 contrôleur qui effectue ce traitement, vous pouvez empaqueter ce code et ainsi le partager.

Les contrôleurs 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 :

2.1.2 Extensions de la Vue

Un *Helper* ou assistant est une classe d'assistance pour les vues. De même que les composants sont utilisés par plusieurs contrôleurs, les assistants permettent à différentes vues d'accéder et de partager une même logique de présentation. L'un des assistants 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 contrôleur ont le même *layout*. Les *elements* sont utilisés lorsque de petites portions de contenu doivent apparaître dans plusieurs vues.

2.1.3 Extensions du Modèle

De même, les Comportements ou *Behaviors* fonctionnent comme des passerelles pour ajouter une fonctionnalité commune aux modèles. Par exemple, si vous stockez les données d'un utilisateur dans une structure en arbre, vous pouvez spécifier que votre modèle 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 modèles sont aussi soutenus par une autre classe nommée une *DataSource* (source de données). Il s'agit d'une couche d'abstraction qui permet aux modèles de manipuler différents types de données de manière consistante. La plupart du temps la source principale de données dans CakePHP est une base de données, vous pouvez cependant écrire des DataSources additionnelles pour représenter des flux RSS, des fichiers CSV, des entrées LDAP ou des évènements iCal. Les *DataSources* vous permettent d'associer des enregistrements issus de sources différentes : plutôt que d'être limité à des jointures SQL, les DataSources vous permettent de dire à votre modèle LDAP qu'il est associé à plusieurs événements iCal.

Tout comme les contrôleurs, les modèles 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 modèles.

2.1.4 Extensions d'application

Contrôleurs, Assistants et Modèles ont chacun une classe parente, que vous pouvez utiliser pour définir des modifications impactant toute l'application. *AppController* (disponible dans /cake/libs/controller/app_controller.php), *AppHelper* (disponible dans /cake/libs/view/helpers/app_helper.php) et *AppModel* (disponible dans /cake/libs/model/app_model.php) sont de bons choix pour écrire les méthodes que vous souhaitez partager entre tous vos contrôleurs, assistants ou modèles.

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 contrôleurs. Le comportement par défaut suppose que l'URL */controller/action/var1/var2* est liée au contrôleur "controller" 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 modèles, de contrôleurs 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.

2.2 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: 2. Requête type CakePHP

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

- Ricardo clique sur le lien pointant vers http://www.example.com/cakes/buy et son navigateur envoie une requête au serveur Web.
- Le routeur analyse l'URL afin d'extraire les paramètres de cette requête : le contrôleur, l'action et tout argument qui affectera la logique métier pendant cette requête.
- En utilisant les routes, l'URL d'une requête est liée à une action d'un contrôleur (une méthode d'une classe contrôleur spécifique). Dans notre exemple, il s'agit de la méthode buy() du Contrôleur Cakes. La fonction de rappel du contrôleur, beforeFilter(), est appelée avant que toute logique de l'action du contrôleur ne soit exécutée.
- Le contrôleur peut utiliser des modèles pour accéder aux données de l'application. Dans cet exemple, le contrôleur utilise un modèle pour récupérer les derniers achats de Ricardo depuis la base de données. Toute méthode de rappel du modèle, tout comportements ou sources de données peuvent s'appliquer pendant cette opération. Bien que l'utilisation du modèle ne soit pas obligatoire, tous les contrôleurs CakePHP nécessitent au départ, au moins un modèle.
- Une fois que le modèle a récupéré les données, elles sont retournées au contrôleur. Des fonctions de rappel du modèle peuvent s'exécuter.
- Le contrôleur peut faire usage de composants pour affiner les données ou pour effectuer d'autres opérations (manipulation de session, authentification ou envoi de mails par exemple).
- Une fois que le contrôleur a utilisé les modèles et composants pour préparer suffisamment les données, ces données sont passées à la vue grâce à la méthode set(). Les méthodes de rappel du contrôleur 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*).
- D'autres fonctions de rappel du contrôleur (comme afterFilter) peuvent être exécutées. La vue complète et final est envoyée au navigateur de Ricardo.

2.3 Structure de fichiers dans CakePHP

Après avoir téléchargé et extrait CakePHP, voici les fichiers et répertoires que vous devriez voir :

- app
- cake
- 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 *cake* 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 coeur 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 répertoire 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.
controllers	Contient vos contrôleurs et leurs composants.
locale	Stocke les fichiers pour l'internationalisation.
models	Pour les modèles, comportements et sources de données de votre application.
plugins	Contient les packages des Plugins.
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 modèles, les logs et parfois les informations de session.
vendors	Toutes classes ou librairies tierces doivent être mises ici, de sorte qu'il sera facile d'y accéder par la fonction vendors(). 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.
views	Les fichiers de présentation sont placés ici : éléments, pages d'erreur, assistants, mises en page 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.

2.4 Conventions 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 cauchemard 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 crées 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.

2.4.1 Conventions pour le nom des fichiers et des classes

En général, les noms de fichiers sont composés avec le caractère souligné (*underscore*), alors que les noms de classe sont *CamelCased*. Donc si vous avez une classe **MaChouetteClasse**, alors dans Cake, le fichier devrait être nommé `ma_chouette_classe.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 Contrôleur **ContrôleurGrosBisous** devrait se trouver dans un fichier nommé `gros_bisous_controller.php` (notez l'ajout de `_controller` dans le nom du fichier)
- La classe Composant (*Component*) **MonSuperComposant** devrait se trouver dans un fichier nommé `mon_super.php`
- La classe Modèle **ModeleValeurOption** devrait se trouver dans un fichier nommé `valeur_option.php`
- La classe Comportement (*Behavior*) **ComportementSpecialementFunkable** devrait se trouver dans un fichier nommé `specialement_funkable.php`
- La classe Vue **VueSuperSimple** devrait se trouver dans un fichier nommé `super_simple.ctp`
- La classe Assistant (*Helper*) **AssistantLeMeilleurQuiSoit** devrait se trouver dans un fichier nommé `le_meilleur_qui_soit.php`

Chaque fichier serait située dans ou sous les répertoires appropriés (qui peuvent être dans un sous-répertoire) de votre répertoire principal App.

2.4.2 Conventions pour les Modèles

Les noms de classe de modèle sont au singulier et *CamelCased*. "Personne", "GrossePersonne" et "VraimentGrossePersonne" en sont des exemples.

Les noms de tables correspondant aux modèles CakePHP sont au pluriel et utilisent le caractère souligné (*underscore*). Les tables correspondantes aux modèles mentionnés ci-dessus seront donc respectivement : "personnes", "grosse_personnes" et "vraiment_grosse_personnes".

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 soulignés (*underscore*) comme ici : nom_de_famille.

Les clés étrangères des relations *hasMany*, *belongsTo* ou *hasOne* sont reconnues par défaut grâce au nom (singulier) du modèle 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 modèles devraient être nommées d'après le nom des tables des modèles qu'elles unissent, dans l'ordre alphabétique ("pommes_zebres" plutôt que "zebres_pommes").

Toutes les tables avec lesquelles les modèles 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, comme les lignes de votre table de jointure "posts_tags", 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 modèle 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.

2.4.3 Conventions pour les Contrôleurs

Les noms des classes de contrôleur sont au pluriel, CamelCased et se terminent par 'Controller'. PersonnesController, GrossePersonnesController et VraimentGrossePersonnesController sont des exemples respectant cette convention.

La première fonction que vous écrivez pour un contrôleur devrait être index(). Lorsqu'une requête adresse un contrôleur mais pas d'action, le comportement par défaut de CakePHP est d'exécuter la fonction index() de ce contrôleur. Ainsi, la requête http://www.exemple.com/pommes/ renvoie à la fonction index() de PommesController, alors que http://www.exemple.com/pommes/view renvoie vers la fonction view() de PommesController.

Dans CakePHP, vous pouvez aussi changer la visibilité des fonctions d'un contrôleur en préfixant le nom par des caractères soulignés. Si une fonction d'un contrôleur a été préfixée avec un souligné, elle ne sera pas visible sur le Web, via le répartiteur, mais elle sera disponible pour un usage interne. Exemple :

```
<?php
class ActualitesController extends AppController {

    function dernieres() {
        $this->_chercherNouvellesActualites();
    }
```

```
    function _chercherNouvellesActualites() {
        //Logique de recherche des dernières actualités
    }
}

?>
```

Alors que la page http://www.exemple.com/actualites/dernieres/ sera accessible pour les utilisateurs, quelqu'un qui essayera cette page http://www.exemple.com/actualites/_chercherNouvellesActualites/ aura une erreur, parce que cette méthode précédé d'un souligné.

2.4.3.1 Considérations sur les URL pour les noms de Contrôleur

Comme vous venez juste de le voir, les noms de contrôleurs en un seul mot se transforment facilement en un simple chemin d'URL en minuscule. Par exemple, le contrôleur `PommesController` (qui devrait être défini dans un fichier nommé 'pommes_controller.php') est accédé par **`http://exemple.com/pommes`**.

Plusieurs mots peuvent être dans des formes 'infléchies' équivalentes au nom du contrôleur comme :

- **`/pommesRouges`**
- **`/PommesRouges`**
- **`/Pommes_rouges`**
- **`/pommes_rouges`**

qui résolveront tous l'index du contrôleur PommesRouges. Cependant, la convention veut que vos urls soit en minuscule et *underscored*, ainsi **`/pommes_rouges/aller_cueillir`** est la forme correcte pour accéder à l'action de contrôleur `PommesRougesController::aller_cueillir`.

Pour plus d'informations sur les URLs CakePHP et la gestion des paramètres, lisez Configuration des Routes.

2.4.4 Conventions pour les Vues

Les fichiers de gabarits de vue (*template*) sont nommés d'après les fonctions du contrôleur qu'elles affichent, sous une forme "soulignée" (*underscored*). La fonction soyezPret() de la classe PersonnesController cherchera un gabarit de vue dans : /app/views/personnes/soyez_pret.ctp

Le schéma classique est "/app/views/contrôleur/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 Modèle : "Personne", trouvée dans /app/models/personne.php
- Classe du Contrôleur : "PersonnesController", trouvée dans /app/controllers/personnes_controller.php
- Gabarit de la Vue : trouvé dans /app/views/personnes/index.ctp

En utilisant ces conventions, CakePHP sait qu'une requête à http://exemple.com/personnes/ sera liée à un appel à la fonction index() du Contrôleur PersonnesController, dans lequel le modèle 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 pour voir comment les choses s'articulent.

3 Développer avec CakePHP

- Un serveur HTTP. Apache avec mod_rewrite est préférable, mais en aucune façon obligatoire.
- PHP 4.3.2 ou plus récent. Oui, CakePHP fonctionne très bien en PHP 4 ou 5.

Techniquement, un moteur de base de données n'est pas obligatoire, mais nous imaginons que la plupart des applications en font usage. CakePHP supporte de nombreux moteurs de bases de données :

- MySQL (4 ou plus)
- PostgreSQL
- Microsoft SQL Server
- Oracle
- SQLite

3.2 Préparation à l'installation

CakePHP est rapide et facile à installer. Les éléments minimum requis sont un serveur web et une copie de Cake, c'est tout ! Bien que ce manuel se concentre essentiellement sur une configuration avec Apache (parce que c'est le plus répandu), vous pouvez configurer Cake pour qu'il fonctionne avec une grande variété de serveurs web, tels LightHTTPD ou Microsoft IIS.

La préparation de l'installation est constituée des étapes suivantes :

- Télécharger une copie de CakePHP
- Configurer, si besoin, votre serveur web pour utiliser PHP
- Vérifier les permissions de fichier

3.2.1 Obtenir CakePHP

Il y a deux méthodes principales pour récupérer une copie récente de CakePHP. Vous pouvez soit télécharger une archive (zip/tar.gz/tar.bz2) sur le site web principal, soit extraire le code depuis le dépôt GIT.

Pour obtenir la dernière version majeure de CakePHP, visitez le site web principal http://www.cakephp.org et suivez le lien "Download Now".

Toutes les versions courantes de CakePHP sont hébergées sur Github. Github héberge à la fois CakePHP lui-même, ainsi que plusieurs autres plugins pour CakePHP. Les versions de CakePHP sont disponibles sur la section téléchargements de Github.

Sinon vous pouvez récupérer le code encore tout chaud, avec toutes les corrections de bug et les améliorations de chaque minute (enfin, de chaque jour). Celles-ci sont accessibles sur Github en clonant le repository. Github.

3.2.2 Droits fichiers

CakePHP utilise le répertoire "/app/tmp" pour un certain nombre d'opérations différentes. Descriptions de modèle, cache de vues et information de session en sont quelques exemples.

Aussi, assurez-vous que ce répertoire soit bien éditable par le serveur web de votre installation Cake.

3.3 Installation

Installer CakePHP peut-être aussi simple que le déposer dans le Document Root de votre serveur web ou bien aussi complexe et souple que vous le souhaitez. Cette section couvrira les 3 types d'installations principaux : 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.

3.3.1 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.exemple.com/cake_1_3/`**. 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_1.3.0). Renommez ce dossier en "cake_1_3". Votre installation "développement" devrait ressembler à quelque chose comme çà dans votre système de fichiers :

- **`/var/www/html`**
  - **`/cake_1_3`**
    - **`/app`**
    - **`/cake`**
    - **`/vendors`**
    - **`/.htaccess`**
    - **`/index.php`**
    - **`/README`**

Si votre serveur web est correctement configuré, vous devriez maintenant accéder à votre application Cake à l'adresse : **`http://www.exemple.com/cake_1_3/`**

3.3.2 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`** (répertoire défini comme le **`DocumentRoot`** du serveur)
  - **`/cake`**
  - **`/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**.

3.3.3 Installation avancée

Il peut y avoir des situations où vous souhaitez placer les répertoires de CakePHP à différents endroits du système de fichiers. Cela peut être du à des restrictions en hébergement mutualisé ou peut-être que vous voulez simplement partager vos librairies Cake entre plusieurs de vos applications. Cette section décrit comment dérouler vos répertoires CakePHP à travers un système de fichiers.

D'abord, sachez qu'il y a 3 parties principales dans une application Cake :

1. Les librairies du coeur de CakePHP, dans **/cake**.
2. Le code de votre application, dans **/app**
3. La racine web de l'application, habituellement dans **/app/webroot**

Chacun de ces répertoires peut être situé n'importe où dans votre système de fichier, à l'exception de webroot, qui doit être accessible via votre serveur web. Vous pouvez même déplacer ce répertoire webroot en dehors du répertoire app, tant que vous dites à Cake où vous l'avez mis.

Pour configurer votre installation Cake, vous aurez besoin de faire quelques modifications dans le fichier **/app/webroot/index.php**. Il y a trois constantes que vous devrez éditer : `ROOT`, `APP_DIR`, et `CAKE_CORE_INCLUDE_PATH`.

Lançons-nous dans un exemple, ainsi vous pourrez voir à quoi doit ressembler, en pratique, une installation avancée. Imaginez que je veuille configurer CakePHP pour fonctionner de la manière suivante :

- Les librairies du coeur de CakePHP seront placées dans **/usr/lib/cake**.
- Le répertoire webroot de mon application sera placé dans **/var/www/monsite/**
- Le répertoire app de mon application sera placé dans **/home/moi/monsite**.

Etant donné ce type de configuration, j'aurais besoin d'éditer mon fichier **/webroot/index.php** (qui se trouvera à **/var/www/monsite/index.php**, dans cet exemple) afin qu'il ressemble à ce qui suit :

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

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

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

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

Il est recommandé d'utiliser la constante **DS** (*Directory Separator*) plutôt que les slashs pour délimiter les chemins de fichier. Cela évite toute erreur de fichier introuvable que vous pourriez avoir en utilisant un mauvais délimiteur et cela rend votre code davantage portable.

3.3.3.1 Chemins de classes additionnels

C'est parfois pratique de pouvoir partager les classes MVC entre applications au sein d'un même système. Si vous voulez le même contrôleur dans 2 applications, vous pouvez utiliser le fichier **bootstrap.php** de CakePHP pour disposer de ces classes additionelles dans une vue.

Dans **bootstrap.php**, définissez quelques variables nommées de façon particulière pour rendre CakePHP conscient des autres emplacements de classes MVC à explorer :

```
App::build(array(
'plugins' => array('/chemin/complet/vers/plugins/',
'/chemin/complet/suivant/vers/plugins/'),
'models' => array('/chemin/complet/vers/models/',
'/chemin/complet/suivant/vers/models/'),
'views' => array('/chemin/complet/vers/views/', '/chemin/complet/suivant/vers/views/'),
'controllers' => array('/chemin/complet/vers/controllers/',
'/chemin/complet/suivant/vers/controllers/'),
'datasources' => array('/chemin/complet/vers/datasources/',
'/chemin/complet/suivant/vers/datasources/'),
'behaviors' => array('/chemin/complet/vers/behaviors/',
'/chemin/complet/suivant/vers/behaviors/'),
'components' => array('/chemin/complet/vers/components/',
'/chemin/complet/suivant/vers/components/'),
'helpers' => array('/chemin/complet/vers/helpers/',
'/chemin/complet/suivant/vers/helpers/'),
'vendors' => array('/chemin/complet/vers/vendors/',
'/chemin/complet/suivant/vers/vendors/'),
'shells' => array('/chemin/complet/vers/shells/',
'/chemin/complet/suivant/vers/shells/'),
'locales' => array('/chemin/complet/vers/locale/',
'/chemin/complet/suivant/vers/locale/')
));
```

L'ordre dans lequel le *bootstrapping* est fait a également changé. Dans le passé, `app/config/core.php` était lu **après** `app/config/bootstrap.php`. Ceci avait pour effet d'empêcher la mise en cache de tous les appels à `App::import()` dans le bootstrap, et ralentissait considérablement l'application. Dans la version 1.3, core.php est chargé et les configurations de mise en cache sont créées **avant** que le bootstrap.php soit chargé.

3.3.4 Apache et mod_rewrite

Bien que CakePHP soit architecturé pour travailler avec *mod_rewrite* tel quel - et il le fait - nous avons remarqué que certains utilisateurs se battent pour obtenir quelque chose qui marche bien sur leurs systèmes.

Voici quelques astuces que vous devriez essayer pour le faire fonctionner correctement. Regardez d'abord votre fichier httpd.conf (Assurez-vous que vous avez édité le httpd.conf du système et non celui d'un utilisateur ou d'un site spécifique).

1. Assurez-vous qu'une réécriture .htaccess est permise et que *AllowOverride* est défini sur *All* pour le *DocumentRoot* adéquat. Vous devriez voir quelque chose comme :

   ```
   # Each directory to which Apache has access can be configured with respect
   # to which services and features are allowed and/or disabled in that
   # directory (and its subdirectories).
   #
   ```

```
# First, we configure the "default" to be a very restrictive set of
# features.
#
<Directory />
    Options FollowSymLinks
    AllowOverride All
#    Order deny,allow
#    Deny from all
</Directory>
```

2. Assurez-vous que vous chargez *mod_rewrite* correctement. Vous devriez voir quelque chose comme :

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

Dans la plupart des systèmes ceci sera commenté (en étant précédé du signe #) par défault, donc vous aurez juste besoin de supprimer ces symboles #.

Après avoir effectué vos modifications, redémarrez Apache pour être sÃ»r que les paramètres seront activés.

Vérifiez que vos fichiers .htaccess sont bien dans les bons répertoires.

Ceci peut arriver pendant la copie car certains systèmes d'exploitation traitent les fichiers commençant '.' comme des fichiers cachés et donc on ne les voit pas après la copie.

3. Assurez-vous que votre copie de CakePHP provient bien de la section téléchargements de ce site ou bien de notre dépôt GIT et qu'elle a été décompressée correctement en vérifiant les fichiers .htaccess.

celui du répertoire principal de Cake (qui a besoin d'être copié dans votre document, ceci redirige tout vers votre application Cake):

```
<IfModule mod_rewrite.c>

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

Celui du répertoire app de Cake (qui sera copié au début du répertoire de votre application par le script bake) :

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

Celui du répertoire webroot de Cake (qui sera copié à la racine web de votre application par le script bake) :

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

Chez la plupart des hébergeurs (GoDaddy, 1and1), votre serveur web est en fait exécuté depuis un répertoire utilisateur qui utilise déjà *mod_rewrite*. Si vous installez CakePHP dans un répertoire utilisateur (http://exemple.com/~pseudo/cakephp/) ou tout autre structure d'URL qui utilise déjà *mod_rewrite*, vous devrez ajouter des déclarations RewriteBase aux fichiers .htaccess que CakePHP utilise (/.htaccess, /app/.htaccess, /app/webroot/.htaccess).

Ceci peut être ajouté dans la même section que la directive RewriteEngine, ainsi, votre fichier .htaccess du webroot devrait ressembler à quelque chose comme çà :

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

Les détails de ces modifications dépendront de votre configuration et pourront inclure des choses supplémentaires qui ne sont pas liées à Cake. Merci de vous référez à la documentation en ligne d'Apache pour plus d'information.

3.3.5 Lighttpd et Pretty_URLs

Bien que Lighttpd propose un module de réécriture, il n'est pas équivalent au mod_rewrite d'Apache. Pour obtenir des 'pretty urls' en utilisant Lighty, vous avez deux possibilités. La première est d'utiliser mod_rewrite, la seconde est d'utiliser un script LUA et mod_magnet.

Avec mod_rewrite

La manière la plus simple pour avoir des 'pretty urls' est d'ajouter ce script dans votre configuration de Lighty. Changez l'URL, et tout devrait bien se passer. Attention ! Ceci ne marche pas lorsque Cake est installé dans un sous-répertoire.

```
$HTTP["host"] =~ "^(www\.)?example.com$" {
        url.rewrite-once = (
                # Cette requête est pour les css|fichiers etc, ne le passer pas dans
Cake
                "/(css|files|img|js)/(.*)" => "/$1/$2",
                "^([^\?]*)(\?(.+))?$" => "/index.php?url=$1&$3",
        )
        evhost.path-pattern = "/home/%2-%1/www/www/%4/app/webroot/"
}
```

Avec mod_magnet

Pour utiliser 'pretty URLs' avec CakePHP et Lighttpd, placez ce script lua dans /etc/lighttpd/cake.

```
-- Une petite fonction assistance
function file_exists(path)
  local attr = lighty.stat(path)
  if (attr) then
      return true
  else
      return false
  end
end
```

```
function removePrefix(str, prefix)
  return str:sub(1,#prefix+1) == prefix.."/" and str:sub(#prefix+2)
end

-- prefix sans le slash
local prefix = ''

-- Magie ! ;)
if (not file_exists(lighty.env["physical.path"])) then
    -- Le fichier est toujours manquant. passez le avec le backend fastcgi
    request_uri = removePrefix(lighty.env["uri.path"], prefix)
    if request_uri then
      lighty.env["uri.path"]          = prefix .. "/index.php"
      local uriquery = lighty.env["uri.query"] or ""
      lighty.env["uri.query"] = uriquery .. (uriquery ~= "" and "&" or "") .. "url=" ..
request_uri
      lighty.env["physical.rel-path"] = lighty.env["uri.path"]
      lighty.env["request.orig-uri"]  = lighty.env["request.uri"]
      lighty.env["physical.path"]     = lighty.env["physical.doc-root"] ..
lighty.env["physical.rel-path"]
    end
end
-- fallthrough va le remettre dans la boucle de requête de Lighty
-- ce qui permet la gestion du message 304 Not Modified. ;)
```

Si vous lancez votre installation CakePHP depuis un sous-répertoire, vous devez paramétrer prefix = 'nom_du_sous_repertoire' dans le script ci-dessus.

Ensuite expliquez à Lighttpd où se trouve votre vhost :

```
$HTTP["host"] =~ "example.com" {
        server.error-handler-404  = "/index.php"

        magnet.attract-physical-path-to = ( "/etc/lighttpd/cake.lua" )

        server.document-root = "/var/www/cake-1.2/app/webroot/"

        # Pensez également à retirer les fichiers tmp de vim
        url.access-deny = (
                "~", ".inc", ".sh", "sql", ".sql", ".tpl.php",
                ".xtmpl", "Entries", "Repository", "Root",
                ".ctp", "empty"
        )
}
```

3.3.6 Jolies URLs avec nginx

nginx est un serveur populaire qui, comme Lighttpd, utilise moins de ressources système. Son inconvénient est qu'il ne fait pas usage des fichiers .htaccess comme Apache et Lighttpd, il est donc nécessaire de créer ces URLs réécrites dans la configuration disponible du site. En fonction de votre paramètrage, vous devrez modifier ceci, mais vous aurez besoin, au minimum, de lancer PHP comme une instance FastCGI.

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

```
server {
    listen   80;
    server_name exemple.com;

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

    location / {
        root   /var/www/exemple.com/public/app/webroot/;
        index  index.php index.html index.htm;
        if (-f $request_filename) {
            break;
        }
        if (-d $request_filename) {
            break;
        }
        rewrite ^(.+)$ /index.php?q=$1 last;
    }

    location ~ .*\.php[345]?$ {
        include /etc/nginx/fcgi.conf;
        fastcgi_pass    127.0.0.1:10005;
        fastcgi_index   index.php;
        fastcgi_param SCRIPT_FILENAME
/var/www/exemple.com/public/app/webroot$fastcgi_script_name;
    }
}
```

3.3.7 URL Rewrites on IIS7 (Windows hosts)

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

IIS7 does not natively support .htaccess files. While there are add-ons that can add this support, you can also import htaccess rules into IIS to use CakePHP's native rewrites. To do this, follow these steps:

1. Use Microsoft's Web Platform Installer to install the URL Rewrite Module 2.0.
2. Create a new file in your CakePHP folder, called **`web.config`**
3. Using Notepad or another XML-safe editor, copy the following code into your new **`web.config`** file...

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
            <rule name="Redirect static resources" stopProcessing="true">
            <match url="^(ico|img|css|files|js)(.*)$" />
            <action type="Rewrite" url="app/webroot/{R:1}{R:2}" appendQueryString="false" />
            </rule>
            <rule name="Imported Rule 1" stopProcessing="true">
            <match url="^(.*)$" ignoreCase="false" />
            <conditions logicalGrouping="MatchAll">
                        <add input="{REQUEST_FILENAME}" matchType="IsDirectory"
negate="true" />
                        <add input="{REQUEST_FILENAME}" matchType="IsFile"
negate="true" />
            </conditions>
            <action type="Rewrite" url="index.php?url={R:1}" appendQueryString="true"
/>
```

```
            </rule>
            <rule name="Imported Rule 2" stopProcessing="true">
              <match url="^$" ignoreCase="false" />
              <action type="Rewrite" url="/" />
            </rule>
            <rule name="Imported Rule 3" stopProcessing="true">
              <match url="(.*)" ignoreCase="false" />
              <action type="Rewrite" url="/{R:1}" />
            </rule>
            <rule name="Imported Rule 4" stopProcessing="true">
              <match url="^(.*)$" ignoreCase="false" />
              <conditions logicalGrouping="MatchAll">
                        <add input="{REQUEST_FILENAME}" matchType="IsDirectory"
negate="true" />
                        <add input="{REQUEST_FILENAME}" matchType="IsFile"
negate="true" />
              </conditions>
              <action type="Rewrite" url="index.php?url={R:1}" appendQueryString="true"
/>
            </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

It is also possible to use the Import functionality in IIS's URL Rewrite module to import rules directly from CakePHP's `.htaccess` files in root, **`/app/`**, and **`/app/webroot/`** - although some editing within IIS may be necessary to get these to work. When Importing the rules this way, IIS will automatically create your **`web.config`** file for you.

Once the **`web.config`** file is created with the correct IIS-friendly rewrite rules, CakePHP's links, css, js, and rerouting should work correctly.

3.3.8 Faites chauffer !

Parfait, voyons CakePHP à l'oeuvre. Selon la configuration que vous avez utilisé, vous devriez pointer votre navigateur web à l'adresse : http://exemple.com/ ou bien : http://exemple.com/installation_cake/. Vous vous trouvez alors en présence de la page d'accueil par défaut de CakePHP et un message vous informe du statut actuel de votre connexion à la base de données.

Félicitations ! Vous êtes prêts à créer votre première application CakePHP.

3.4 Configuration

Configurer une application CakePHP c'est du gâteau. Après que vous ayez installé CakePHP, créer 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 avantage de l'architecture flexible de CakePHP. Vous pouvez facilement ajouter des fonctionnalités héritant du coeur de CakePHP, configurer des URLs additionnelles/différentes (routes) et définir des inflexions additionnelles/différentes.

3.4.1 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 :

```
var $default = array('driver'      => 'mysql',
                     'persistent'  => false,
                     'host'          => 'localhost',
                     'login'         => 'cakephputilisateur',
                     'password'   => 'c4k3roxx!',
                     'database'   => 'ma_base_cakephp',
                     'prefix'        => '');
```

Le tableau de connexion $default est utilisé tant qu'aucune autre connexion n'est spécifiée dans un modèle, 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 modèles, en créant un nouveau tableau de connexion à la base de données, intitulé $ancienne, identique au tableau $default, puis en initialisant la propriété $useDbConfig = 'ancienne' dans les modèles appropriés.

Complétez les couples clé/valeur du tableau de configuration pour répondre au mieux à vos besoins.

Clé	Valeur
driver	Le nom du pilote de base de données pour lequel ce tableau est destiné. Exemples : mysql, postgres, sqlite, pear-drivername, adodb-drivername, mssql, oracle, or odbc.
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 (optionnel)	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 (optionnel)	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'

Le paramétrage du préfixe est valable pour les tables, **pas** pour les modèles. Par exemple, si vous créez une table de liaison entre vos modèles Pomme et Saveur, vous la nommerez "prefixe_pommes_saveurs" (et non pas "prefixe_pommes_prefixe_saveurs") et vous paramétrerez votre propriété "prefix" sur "prefixe_".

A présent, vous aurez peut-être envie de jeter un oeil aux Conventions 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 grosse_boites, votre modèle GrosseBoite, votre contrôleur GrosseBoitesController, tout marchera ensemble automatiquement. Par

convention, utilisez les *underscores*, les minuscules et les formes plurielles pour les noms de vos tables - par exemple : cuisiniers, magasin_de_pates et bon_biscuits.

3.4.2 Configuration du coeur de Cake

La configuration de l'application dans CakePHP se trouve dans /app/config/core.php. Le fichier est une collection de définitions de variables de la classe Configure et de définitions de constantes qui déterminent comment votre application se comporte. Avant de nous plonger dans l'étude de ces variables particulières, vous devrez vous familiariser avec "Configure", la classe de configuration centrale de CakePHP.

3.4.3 La classe Configuration

Malgré quelques trucs nécessitant d'être configurés dans CakePHP, c'est souvent utile d'avoir vos propres règles de configuration pour votre application. Auparavant, vous deviez définir des valeurs de configuration personnalisées en déclarant des variables ou des constantes dans quelques fichiers. Cela vous obligeait à inclure ces fichiers de configuration chaque fois que vous aviez besoin d'utiliser ces valeurs.

La nouvelle classe Configure de CakePHP peut être utilisée pour stocker et récupérer des valeurs spécifiques à votre application ou à l'exécution. Attention, cette classe permet de stocker n'importe quoi en son sein, pour l'utiliser dans tout autre partie du code : c'est une tentation évidente de casser le motif MVC pour lequel CakePHP est architecturé. L'objectif principal de la classe Configure est de conserver des variables centralisées, qui peuvent être partagées entre plusieurs objets. Souvenez-vous d'essayer de fonctionner sur le mode "convention plutôt que configuration" et vous ne risquerez pas de casser la structure MVC que nous avons mise en place.

Cette classe agit comme un singleton et ses méthodes peuvent être appelées n'importe où dans votre application, dans un contexte statique.

```
<?php Configure::read('debug'); ?>
```

3.4.3.1 Méthodes de la classe Configure

```
write(string $key, mixed $value)
```

Utilisez write() pour stocker une donnée dans la configuration de l'application.

```
Configure::write('Societe.nom','Pizza');
Configure::write('Societe.slogan','Pizza pour votre corps et votre âme');
```

Remarquez l'utilisation de la notation avec point pour le paramètre `$key`. Vous pouvez utiliser cette notation pour organiser votre configuration en groupes logiques.

L'exemple ci-dessus aurait pu s'écrire aussi en un simple appel :

```
Configure::write(
    'Societe',array('nom'=>'Pizza','slogan'=>'Pizza pour votre corps et votre
âme')
);
```

Vous pouvez utiliser `Configure::write('debug', $int)` pour passer du mode débug au mode production à la volée. Ceci est particulièrement pratique pour les interactions AMF ou SOAP dans lesquelles les informations de débug peuvent poser des problèmes de *parsing*.

3.4.3.1.2 read

`read(string $key = 'debug')`

Utilisé pour lire les données de configuration de l'application. Mis par défaut à la valeur importante 'debug' de CakePHP. Si une clé est passée, la données correspondante est retournée. En reprenant notre exemple de la fonction write() ci-dessus, nous pouvons lire cette donnée en retour :

```
Configure::read('Societe.nom');     // retourne : 'Pizza'
Configure::read('Societe.slogan'); // retourne : 'Pizza pour votre corps et votre
âme'

Configure::read('Societe');

// retourne :
array('nom' => 'Pizza', 'slogan' => 'Pizza pour votre corps et votre âme');
```

3.4.3.1.3 delete

`delete(string $key)`

Utilisez cette méthode pour supprimer des informations de configuration.

```
Configure::delete('Societe.nom');
```

3.4.3.1.4 load

`load(string $path)`

Utilisez cette méthode pour charger des informations de configuration depuis un fichier spécifique.

```
// /app/config/messages.php:
<?php
$config['Societe']['nom'] = 'Pizza';
$config['Societe']['slogan'] = 'Pizza votre corps et votre âme';
$config['Societe']['telephone'] = '01-02-03-04-05';
?>

<?php
Configure::load('messages');
Configure::read('Societe.nom');
?>
```

Chaque paire clé-valeur est représentée dans le fichier par le tableau `$config`. Toute autre variable dans le fichier sera ignorée par la méthode `load()`.

3.4.3.1.5 version

```
version()
```

Retourne la version de CakePHP utilisée par l'application courante.

3.4.3.2 Variables de 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 des effets que leur utilisation entraîne pour votre application CakePHP.

Variable de Configure	Description
debug	Modifie la sortie de debug CakePHP. 0 = Mode production. Pas de sortie. 1 = Montre les erreurs et les alertes. 2 = Montre les erreurs, les alertes et le SQL.
App.baseUrl	Décommentez cette définition si vous **ne prévoyez pas** d'utiliser le mod_rewrite d'Apache avec CakePHP. N'oubliez pas de supprimer également vos fichiers .htaccess.
Routing.prefixes	Décommentez cette définition si vous aimeriez tirer profit des routes préfixées de CakePHP comme admin. Définissez cette variable comme un tableau des noms des préfixes de routes que vous souhaiteriez utiliser. Plus d'informations sur celà ultérieurement.
Cache.disable	Quand il est réglé à true, le cache est désactivé pour l'ensemble du site.
Cache.check	Si réglé à true, active le cache de vue. L'activation est encore requise dans les contrôleurs, mais cette variable permet la détection de ces paramètres.
Session.save	Indique à CakePHP quel mécanisme de stockage des sessions utiliser. php = Utiliser le stockage de session PHP par défaut. cache = Utiliser le moteur de cache configuré par Cache::config(). Très utile en conjonction avec Memcache (dans une configuration comportant plusieurs serveurs) afin de stocker à la fois les données et sessions du cache. cake = Stocker les sessions dans /app/tmp database = Stocker les données de session dans une table de la base de données. Assurez-vous de configurer correctement la table en utilisant le fichier SQL situé dans /app/config/sql/sessions.sql.
Session.table	Le nom de la table (sans inclure aucun préfixe) qui enregistre les informations de session.
Session.database	Le nom de la base de données qui enregistre les informations de session.
Session.cookie	Le nom du cookie utilisé pour tracer les sessions.
Session.timeout	Base du temps de déconnexion de la session, en secondes. La valeur réelle dépend du paramètre Security.level.
Session.start	Démarre automatiquement les sessions quand réglé à true.
Session.checkAgent	Quand réglé à false, les sessions CakePHP n'effectueront pas d'analyse pour

	s'assurer que l'agent utilisateur ne change pas entre les requêtes.
Security.level	Le niveau de sécurité CakePHP. Le temps de déconnexion de la session, défini par le paramètre 'Session.timeout', est multiplié par le paramètre indiqué ici. Valeurs possibles : 'high' = x 10 'medium' = x 100 'low' = x 300 'high' et 'medium' active également session.referer_check Les IDs de session CakePHP sont aussi regénérés entre les requêtes si 'Security.level' est défini à 'high'.
Security.salt	Une chaîne aléatoire utilisée par le hash de sécurité.
Asset.timestamp	Ajoute le timestamp de la dernière modification du fichier à la fin des urls fichiers ressources (CSS, JavaScript, Image) quand les bons helpers sont utilisés. Valeurs possibles: (bool) false - Rien n'est fait (par défaut) (bool) true - Ajoute le timestamp quand debug > 0 (string) 'force' - Ajoute le timestamps quand debug >= 0
Acl.classname, Acl.database	Constantes utilisées par les fonctionnalités de Listes de Contrôle d'Accès (*Access Control List - ACL*) de CakePHP. Voyez le chapitre sur les ACL pour plus d'informations.

La configuration du cache se trouve aussi dans le fichier core.php - Nous le couvrirons plus tard, donc restez à l'écoute.

La classe Configure peut être utilisée pour lire et écrire des paramètres de configuration du coeur à la volée. Ceci est particulièrement pratique si vous voulez, par exemple, activer des paramètres de debug pour une section limitée de votre logique applicative.

3.4.3.3 Constantes de configuration

Alors que la plupart des options de configuration sont prises en charge par Configure, il y a quelques constantes que CakePHP utilise à l'exécution.

Constante	Description
LOG_ERROR	Constante d'erreur. Utilisée pour différencier les erreurs enregistrées et les erreurs de débug. Actuellement PHP supporte la constante LOG_DEBUG.

3.4.4 La classe App

Le chargement de classes additionnelles est devenu plus simple dans CakePHP. Dans les versions précédentes, il y avait plusieurs fonctions pour charger une classe nécessaire, basées sur les type de classe que vous souhaitiez charger. Ces fonctions sont devenues obsolètes, toute classe ou librairie devrait maintenant pouvoir être chargée normalement avec App::import(). App::import() s'assure qu'une classe n'est chargée qu'une fois, que la classe parente appropriée a été chargée et résout automatiquement les chemins dans la plupart des cas.

3.4.4.1 Utiliser App::import()

```
App::import($type, $name, $parent, $search, $file, $return)
```

A première vue, `App::import` semble complexe, pourtant, dans la plupart des cas d'utilisation, seuls 2 arguments sont requis.

3.4.4.2 Importer les librairies du coeur

Les librairies du cœur de Cake, comme Sanitize et Xml peuvent être chargées via :

```
<?php App::import('Core', 'Sanitize') ?>
```

Ceci rendra la classe Sanitize disponible.

3.4.4.3 Importer des Contrôleurs, des Modèles, des Composants, des Comportements et des Assistants

Toute classe relative à l'application devrait aussi être chargée avec App::import(). Les exemples suivants illustrent comment s'y prendre.

3.4.4.3.1 Charger des Contrôleurs

```
App::import('Controller', 'MonController');
```

Appeler App::import est équivalent à l'inclusion, par require, du fichier. Il est important de comprendre que la classe nécessite, par la suite, d'être initialisée.

```
<?php
// Identique à : require('controllers/utilisateurs_controller.php');
App::import('Controller', 'Utilisateurs');

// Nous avons besoin de charger la classe
$Users = new UsersController;

// Si nous voulons que les associations de modèle, les components, etc. soient
chargés
$Users->constructClasses();
?>
```

3.4.4.3.2 Charger des Modèles

```
App::import('Component', 'Auth');
```

3.4.4.3.4 Charger des Comportements

```
App::import('Helper', 'Html');
```

3.4.4.3.6 Charger des Assistants

Charger des classes de plugins fonctionne à peu près de la même manière que le chargement des classes du cœur ou de l'application, excepté que vous devez préciser le plugin depuis lequel vous chargez.

```
App::import('Model', 'MonPlugin.Pomme');
```

Pour charger APP/plugins/mon_plugin/vendors/flickr/flickr.php

```
App::import('Vendor', 'MonPlugin.flickr/flickr');
```

3.4.4.5 Charger les fichiers Vendor

La fonction vendor() a été dépréciée. Les fichiers Vendor devraient maintenant être chargés via App::import(). La syntaxe et les arguments additionnels sont légèrement différents, car les structures du fichier vendor peuvent différer grandement et tous les fichiers vendor ne contiennent pas de classes.

Les exemples suivants illustrent comment charger les fichiers vendor depuis divers structures de chemins. Ces fichiers vendor pourraient être localisés dans n'importe quels dossiers de vendor.

3.4.4.5.1 Exemples Vendor

Pour charger **vendors/geshi.php**

```
App::import('Vendor', 'geshi');
```

Pour charger **vendors/flickr/flickr.php**

```
App::import('Vendor', 'flickr/flickr');
```

Pour charger **vendors/un.nom.php**

```
App::import('Vendor', 'UnNom', array('file' => 'un.nom.php'));
```

Pour charger **vendors/services/bien.dit.php**

```
App::import('Vendor', 'BienDit', array('file' => 'services'.DS.'bien.dit.php'));
```

Cela ne fera aucune différence si vos fichiers vendor sont dans votre répertoire /app/vendors. Cake les trouvera automatiquement.

Pour charger **app/vendors/nomVendeur/libFichier.php**

```
App::import('Vendor', 'unIdentifiantUnique', array('file'
=>;'nomVendeur'.DS.'libFichier.php'));
```

3.4.5 Configuration des Routes

Avant d'apprendre comment configurer vos propres routes, vous devriez savoir que CakePHP arrive configuré avec un ensemble de routes par défaut. Le routage par défaut de CakePHP vous emmènera assez loin dans toute application. Vous pouvez accéder à une action directement via l'URL en insérant son nom dans la requête. Vous pouvez aussi passer des paramètres aux action de votre contrôleur en utilisant l'URL.

```
    URL motif de routes par défaut :
    http://exemple.com/controleur/action/param1/param2/param3
```

L'URL /posts/voir correspond à l'action voir() du contrôleur PostsController et /produits/voir_autorisation correspond à l'action voir_autorisation() du contrôleur ProduitsController. Si aucune action n'est spécifiée dans l'URL, la méthode index() est sous-entendue.

La configuration du routage par défaut vous permet également de passer des paramètres aux actions en utilisant l'URL. Par exemple, une requête pour /posts/voir/25 serait équivalente à l'appel de voir(25) dans le contrôleur PostsController.

3.4.5.2 Arguments passés

Les arguments passés sont des arguments additionnels ou des segments du chemin qui sont utilisés lors d'une requête. Ils sont souvent utilisés pour transmettre des paramètres aux méthodes de vos contrôleurs.

```
http://localhost/calendriers/voir/recents/mark
```

Dans l'exemple ci-dessus, `recents` et `mark`sont tous deux des arguments passés à `CalendriersController::voir()`. Les arguments passés sont transmis aux contrôleurs de deux manières. D'abord comme arguments de la méthode de l'action appelée, mais aussi en étant accessibles dans `$this->params['pass']` sous la forme d'un tableau indexé numériquement. 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.

3.4.5.3 Paramètres nommés

Vous pouvez nommer les paramètres et envoyer leurs valeurs en utilisant l'URL. Une requête pour "/posts/voir/titre:premier+post/categorie:general" résultera en un appel à l'action "voir()" du contrôleur PostsController. Dans cette action, vous trouverez les valeurs des paramètres "titre" et "categorie" respectivement dans $this->passedArgs['titre'] et $this->passedArgs['categorie']. Vous pouvez également accéder aux paramètres nommés depuis `$this->params['named']`. `$this->params['named']` contient un tableau de paramètres nommés indexés par leurs nom.

Quelques exemples de routes par défaut seront plus parlants.

```
URL correspondant à une action de contrôleur en utilisant les routes par défaut :

URL : /singes/saute
Correspond à : SingesController->saute();

URL : /produits
Correspond à : ProduitsController->index();

URL: /taches/voir/45
Correspond à : TachesController->voir(45);
```

```
URL: /donations/voir/recentes/2001
Correspond à : DonationsController->voir('recentes', '2001');

URL: /contenus/voir/chapitre:modeles/rubrique:associations
Correspond à : ContenusController->voir();
$this->passedArgs['chapitre'] = 'modeles';
$this->passedArgs['rubrique'] = 'associations';
$this->params['named']['chapitre'] = 'modeles';
$this->params['named']['rubrique'] = 'associations';
```

Lorsque l'on fait des routes personnalisées, un piège classique est d'utiliser des paramètres nommés qui casseront vos routes. Pour résoudre celà vous devez informer le Router des paramètres qui sont censés être des paramètres nommés. Sans cette information le Router 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()`.

3.4.5.4 Définir des Routes

Définir vos propres routes vous permet de déterminer comment votre application répondra à une URL donnée. Définissez vos propres routes dans le fichier **`/app/config/routes.php`** en utilisant la méthode `Router::connect()`.

La méthode `connect()` prend jusqu'à trois paramètres : l'URL que vous souhaitez détecter, les valeurs par défaut pour les éléments personnalisés de la route et des règles à base d'expression régulière pour aider le routeur à trouver les éléments dans l'URL.

Le format de base pour la définition d'une route est :

```
Router::connect(
    'URL',
    array('nomParametre' => 'valeur_par_defaut'),
    array('nomParametre' => 'expression_a_detecter')
)
```

Le premier paramètre est utilisé pour indiquer au routeur quelle sorte d'URL vous essayez de contrôler. L'URL est une chaîne normale délimitée par des *slash*, mais elle peut aussi contenir un joker (*) ou des éléments de route personnalisés (noms de variables préfixés par deux points). L'utilisation d'un joker indique au routeur quels types d'URLs vous voulez détecter et en spécifiant des éléments de route cela vous permet de rassembler les paramètres pour les actions de vos contrôleurs.

Une fois que vous avez spécifié une URL, vous utilisez les deux derniers paramètres de `connect()` pour indiquer à CakePHP que faire avec une requête une fois qu'elle a été détectée. Le second paramètre est un tableau associatif. Les clés du tableau devraient être nommées d'après les éléments de route dans l'URL ou d'après les éléments par défauts ":controller", ":action" et ":plugin". Les valeurs du tableau sont les valeurs par défaut pour ces clés. Regardons quelques exemples simples avant de commencer à utiliser le troisième paramètre de `connect()`.

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

Cette route se trouve dans le fichier routes.php distribué avec CakePHP (ligne 40). Cette route intercepte toute URL commençant par **/pages/** et la transmet à la méthode `display()` du contrôleur `PagesController();`. La requéte **/pages/produits** devrait correspondre à `PagesController->display('produits')`, par exemple.

```
Router::connect(
    '/gouvernement',
    array('controller' => 'produits', 'action' => 'display', 5)
);
```

Ce second exemple montre comment vous pouvez utiliser le deuxième paramètre de `connect()` pour définir des paramètres par défaut. Si vous construisez un site qui propose des produits pour différentes catégories de consommateurs, vous pourriez envisager de créer une route. Cela vous permet d'avoir un lien vers **/gouvernement** plutôt que vers **/produits/display/5**.

Une autre utilisation courante du routeur est de définir un "alias" pour un contrôleur. Disons qu'au lieu d'accéder à notre URL régulière **/utilisateurs/une_action/5**, nous aimerions être en mesure d'y accéder par **/cuisiniers/une_action/5**. La route suivante s'occupe facilement de cela :

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

Ceci indique au routeur que toute URL commençant par **/cuisiniers/** devra être envoyée au contrôleur "utilisateurs".

Quand on génère des urls, les routes sont utilisées aussi. Utiliser `array('controller' => 'utilisateurs', 'action' => 'uneAction', 5)` comme une url affichera **/cuisiniers/uneAction/5** si la route ci-dessus est la première trouvée.

Si nous comptons utiliser des paramètres personnalisés dans notre route, il faut le spécifier au routeur en utilisant la fonction `Router::connectNamed`. Donc si vous voulez la route spécifiée plus haut de reconnaître des URL comme `/cuisiniers/uneAction/type:chef`, nous devons:

```
Router::connectNamed(array('type'));
Router::connect(
    '/cuisiniers/:action/*', array('controller' => 'utilisateurs', 'action' =>
'index')
);
```

Pour plus de flexibilité, vous pouvez spécifier des éléments personnalisés de route. Faire cela vous donne le pouvoir de définir les positions des paramètres dans l'URL pour qu'ils correspondent à ceux des actions du contrôleur. Quand une requête est faite, les valeurs pour ces éléments de route personnalisés se trouvent dans la variable $this->params du contrôleur. Ceci est différent de la façon dont sont traités les paramètres nommés, notez la distinction : les paramètres nommés (/controller/action/nom:valeur) se trouve dans $this->passedArgs, alors que les données des éléments personnalisés de route se trouve dans $this->params. Quand vous définissez un élément personnalisé de route, vous devez aussi spécifier une expression régulière - cela indique à CakePHP comment savoir si l'URL est correctement formée ou pas.

```
Router::connect(
    '/:controller/:id',
```

```
    array('action' => 'voir'),
    array('id' => '[0-9]+')
);
```

Ce simple exemple illustre comment créer une voie rapide pour voir les modèles depuis tout contrôleur, en façonnant une URL qui ressemble à /nom_controleur/id. L'URL fournie à connect() spécifie deux éléments de route : ":controller" et ":id". L'élément ":controller" est un élément de route par défaut de CakePHP, donc le routeur sait comment intercepter et identifier les noms de contrôleur dans les URLs. L'élément ":id" est élément de route personnalisé et doit être davantage clarifier en spécifiant une expression régulière détectable dans le troisième paramètre de connect(). Cela indique à CakePHP comment reconnaître l'ID dans l'URL par opposition à tout autre chose, comme par exemple le nom d'une action.

Une fois que cette route a été définie, requêter /pommes/5 est la même chose que /pommes/voir/5. Les deux appelleront la méthode voir() du contrôleur PommesController. A l'intérieur de la méthode voir(), vous aurez besoin d'accéder à l'ID passé par $this->params['id'].

Encore un exemple et vous serez un pro du routage.

```
Router::connect(
    '/:controller/:annee/:mois/:jour',
    array('action' => 'index', 'jour' => null),
    array(
        'annee' => '[12][0-9]{3}',
        'mois' => '0[1-9]|1[012]',
        'jour' => '0[1-9]|[12][0-9]|3[01]'
    )
);
```

Ceci est plutôt compliqué, mais montre comment les routes peuvent vraiment devenir puissantes. L'URL soumise a 4 éléments de route. Le premier nous est familier : c'est un élément de route par défaut qui indique à CakePHP d'attendre un nom de contrôleur.

Ensuite, nous spécifions quelques valeurs par défaut. Sans tenir compte du contrôleur, nous voulons appeler l'action index(). Nous définissons le paramètre jour à null (le quatrième élément dans l'URL) pour signaler qu'il est optionnel.

Finalement, nous spécifions quelques expressions régulières qui correspondront aux années, mois et jours sous forme numérique. Notez que les parenthèses (groupements) ne sont pas supportées dans les expressions régulières. Vous pouvez quand même en spécifier d'autres, comme ci-dessus, mais ne les groupez pas avec des parenthèses.

Une fois définie, cette route détectera /articles/2007/02/01, /posts/2004/11/16 et /produits/2001/05 (tel que défini, le paramètre jour est optionnel car il a une valeur par défaut), transmettant les requêtes aux actions index() de leurs contrôleurs respectifs, avec les paramètres personnalisés de date dans $this->params.

3.4.5.5 Passer des paramètres à une action

Considérons que votre action a été définie comme ceci et que vous voulez accéder aux arguments en utilisant `$articleID` plutôt que `$this->params['id']`, ajoutez simplement un tableau supplémentaire dans le 3eme paramètre de `Router::connect()`.

```
// un_controller.php
function voir($articleID = null, $slug = null) {
```

```
    // un peu de code ici...
}

// routes.php
Router::connect(
    // Exemple /blog/3-CakePHP_Rocks
    '/blog/:id-:slug',
    array('controller' => 'blog', 'action' => 'voir'),
    array(
        // rien de compliqué, puisque ceci fera simplement correspondre ":id"
à $articleID dans votre action
        'pass' => array('id', 'slug'),
        'id' => '[0-9]+'
    )
)
```

Et maintenant, merci aux capacités de routage inverse, vous pouvez passer dans le tableau d'url comme ci-dessous et Cake saura comment formé l'URL telle que définie dans les routes.

```
// voir.ctp
// ceci retourne un lien vers /blog/3-CakePHP_Rocks
<?= $html->link('CakePHP Rocks', array(
    'controller' => 'blog',
    'action' => 'voir',
    'id' => 3,
    'slug' => Inflector::slug('CakePHP Rocks')
)) ?>
```

3.4.5.6 Préfixe 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/utilisateurs/editer/5`**. Dans CakePHP, le routage admin peut être activé depuis le fichier de configuration du coeur en réglant le chemin d'admin via Routing.admin.

```
Configure::write('Routing.admin', 'admin');
```

Dans votre contrôleur, toute action avec le préfixe `admin_` sera appelée. En utilisant notre exemple des utilisateurs, accéder à l'url **`/admin/utilisateurs/editer/5`** devrait appeler la méthode `admin_editer` de notre `UtilisateursController` en passant 5 comme premier paramètre. Le fichier de vue correspondant devra être **`app/views/utilisateurs/admin_editer.ctp`**

Vous pouvez faire correspondre l'url **`/admin`** à votre action `admin_index` du contrôleur Pages en utilisant la route suivante :

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

Vous pouvez aussi vous servir du Router pour utiliser des préfixes à destination d'autres choses que du routage d'admin.

```
Router::connect('/profils/:controller/:action/*', array('prefix' => 'profils',
```

```
'profils' => true));
```

Tout appel à la section "profils" devrait entraîner la recherche du préfixe `profils_` dans les appels de méthode. Notre exemple d'utilisateurs aurait une structure d'url qui ressemble à **`/profils/utilisateurs/editer/5`** qui appelerait la méthode `profils_editer` du `UtilisateursController`. Une autre chose importante à retenir, l'utilisation du helper HTML pour construire vos liens aidera à maintenir les appels de préfixe. Voici construire ce lien en utilisant le *helper* HTML :

```
echo $html->link('Editer votre profil', array('profils' => true, 'controller' =>
'utilisateurs', 'action' => 'editer', 'id' => 5));
```

Vous pouvez définir plusieurs routes préfixées en utilisant cette approche afin de créer une structure d'URL flexible pour votre application.

3.4.5.7 Routage 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('Nouveau todo', array('plugin' => 'todo', 'controller' =>
'todo_items', 'action' => 'creer'));
```

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('Nouveau todo', array('plugin' => null, 'controller' =>
'utilisateurs', 'action' => 'profil'));
```

En définissant `plugin => null`, vous indiquez au Routeur que vous souhaitez créer un lien qui n'est pas une partie d'un plugin.

3.4.5.8 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/:titre',
        array('controller' => 'pages', 'action' => 'voir'),
```

```
        array(
            'pass' => array('titre')
        )
    );
```

Ensuite pour créer des liens qui s'adapteront aux routes utilisez simplement :

```
$html->link('Titre du lien', array('controller' => 'pages', 'action' => 'voir',
'titre' => Inflector::slug('texte à transformer', '-'), 'ext' => 'html'))
```

3.4.5.9 Classes de route personnalisés

Les classes de route personnalisés vous permettent d'étendre et de modifier la façon dont certaines routes demandes d'analyser et de traiter des routes inversés. Une classe de la route devrait hériter de la classe `CakeRoute` et mettre en oeuvre un ou des deux `match ()` et `parse ()` . Parse est utilisée pour analyser les demandes et correspondance et il est utilisée pour traiter les routes inversés.

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 la classe `SlugRoute` et vous permettent de mettre en oeuvre la gestion des paramètres personnalisés

3.4.6 Inflexions personnalisées

Les conventions de nommage de Cake peuvent être vraiment sympas. Vous pouvez nommer votre table de base de données *big_boxes*, votre modèle *BigBox*, votre controleur *BigBoxesController* et tout fonctionne ensemble automatiquement. La manière dont CakePHP s'y prend pour savoir comment relier les choses ensemble, consiste à infléchir les mots entre leurs formes singulier et pluriel.

Dans certaines occasions (spécialement pour nos amis qui ne parlent pas anglais), vous pouvez rencontrer des situations où "l'inflecteur" de CakePHP (la classe qui "pluralise", "singularise", *camelCase*, et "sous_ligne") ne fonctionnera pas comme vous le souhaiteriez. Si CakePHP ne veut pas reconnaître votre *Foci* (Ndt : masculin pluriel, des foyers, en mathématique/physique) ou votre *Fish*, éditer le fichier de configuration des inflexions est la chose à faire, pour lui expliquer vos cas particuliers. Ce fichier se trouve dans **`/app/config/inflections.php`**.

Dans ce fichier vous trouverez six variables. Chacune vous permet de définir finement le comportement d'inflexion de CakePHP.

Variable du fichier "inflections.php"	Description
$pluralRules	Ce tableau contient les expression régulières pour les cas particuliers de mise au pluriel. Les clés du tableaux sont les motifs et les valeurs les correspondances.

$uninflectedPlural	Un tableau contenant des mots qui ne nécessitent pas d'être modifiés pour passer au pluriel.
$irregularPlural	Un tableau contenant des mots et leur pluriel. Les clés contiennent la forme singulier, les valeurs la forme plurielle. Ce tableau devrait être utilisé pour stocker des mots qui ne suivent pas les règles définies dans $pluralRules.
$singularRules	Identique à $pluralRules, ce tableau regroupe les règles qui "singularisent" les mots.
$uninflectedSingular	Identique à $uninflectedPlural, ce tableau regroupe les mots qui n'ont pas de singulier. Il est égal à $uninflectedPlural par défaut.
$irregularSingular	Identique à $irregularPlural, pour les mots au singulier.

3.4.7 L'amorçage de CakePHP

Si vous avez des besoins additionnels de configuration, utilisez le fichier d'amorçage de CakePHP, qui se trouve dans **`/app/config/bootstrap.php`**. Ce fichier est exécuté juste après le processus d'amorçage du coeur de Cake.

Ce fichier est idéal pour un certain nombre de tâches courantes à lancer au démarrage :

- Définir des fonctions de confort
- Enregistrer des constantes globales
- Définir des chemins additionnels de modèles, vues et contrôleurs

Assurez-vous de maintenir le motif de conception logiciel MVC quand vous ajoutez des choses dans le fichier d'amorçage : il peut être tentant de placer ici des fonctions de formattages au lieu de les utiliser dans vos contrôleurs.

Résistez à la tentation. Plus tard, vous ne regrettez pas de l'avoir fait.

Vous pouvez aussi envisager de déposer des choses dans la classe AppController. Cette classe est la classe parente de tous les contrôleurs de votre application. AppController est un endroit pratique pour utiliser les méthodes de rappel (*callbacks*) et définir des méthodes utilisables par tous vos contrôleurs.

3.5 Contrôleurs

Un contrôleur est utilisé pour gérer la logique d'une partie de votre application. En règle générale, les contrôleurs sont utilisés pour gérer la logique d'un seul modèle. Par exemple, si vous construisiez un site pour une boulangerie en ligne, vous auriez sÃ»rement un contrôleur "Recettes" et un contrôleur "Ingredients" vous permettant de gérer vos recettes et leurs ingrédients. Dans CakePHP, les contrôleurs portent le nom de leur modèle correspondant, au pluriel.

Le modèle Recette est pris en charge par le Contrôleur Recettes, le modèle Produit par le Contrôleur Produits, et ainsi de suite.

Les contrôleurs de votre application sont des classes qui étendent la classe CakePHP AppController, qui hérite elle-même de la classe Controller du coeur. La classe AppController peut être définie dans **`/app/app_controller.php`** et elle devrait contenir les méthodes partagées par tous les contrôleurs de votre application. Elle étend la classe Controller qui est une librairie standard de CakePHP.

Les contrôleurs peuvent inclure un nombre quelconque de méthodes qui sont généralement appelées *actions*. Les actions sont les méthodes du contrôleur utilisées pour afficher des vues. Une action correspond à une

seule méthode d'un contrôleur.

Le *dispatcher* (répartiteur) de CakePHP appelle ces actions quand une requête entrante fait correspondre une URL à une action de contrôleur (reportez-vous à la section "Configuration des routes", pour des explications sur la manière dont les actions de contrôleur et leurs paramètres sont mappés d'après l'URL).

Si l'on se réfère ainsi à notre exemple précédent de boulangerie en ligne, notre contrôleur "Recettes" contiendra sÃ»rement les actions "voir()", "partager()" et "rechercher()". Ce contrôleur devrait se trouver dans **`/app/controllers/recettes_controller.php`** et contenir :

```
# /app/controllers/recettes_controller.php
<?php
    class RecettesController extends AppController {
        function voir($id) {
            //on insérera ici la logique propre à l'action
        }

        function partager($client_id, $recette_id) {
             //on insérera ici la logique propre à l'action
        }

        function rechercher($requete) {
             //on insérera ici la logique propre à l'action
        }
    }

?>
```

Afin d'utiliser tout le potentiel d'un contrôleur dans votre application, nous allons aborder ici quelques-uns des principaux attributs et méthodes offerts par les contrôleurs de CakePHP.

3.5.1 Le contrôleur App

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

```
<?php
class AppController extends Controller {
}
?>
```

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

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 contrôleurs sont fournis, comme la liste des composants ou assistants utilisés par un contrôleur. Dans ces situations, les valeurs des tableaux de AppController sont fusionnées avec les tableaux de la classe contrôleur enfant.

CakePHP fusionne les variables suivantes de la classe AppController avec celles des contrôleurs de votre application :

- $components
- $helpers
- $uses

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

Pensez à appeler les fonctions de retour (*callbacks*) de AppController dans celles du contrôleur enfant pour de meilleurs résultats :

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

3.5.2 Le contrôleur "Pages"

Le coeur de CakePHP est livré avec un contrôleur par défaut appelé *Pages Controller* (**`cake/libs/controller/pages_controller.php`**). La page d'accueil que vous voyez juste après l'installation est d'ailleurs générée à l'aide de ce contrôleur. Il est généralement utilisé pour diffuser des pages statiques. Ex : Si vous écrivez un fichier de vue **`app/views/pages/a_propos.ctp`**, vous pouvez y accéder en utilisant l'url **`http://example.com/pages/a_propos`**

Quand vous **`"cuisinez"`** une applications avec l'utilitaire console de CakePHP, le contrôleur Pages est copié dans votre dossier **`app/controllers/`** et vous pouvez le modifier selon vos besoin. Ou vous pouvez simplement copier le fichier **`pages_controller.php`** du coeur de CakePHP vers votre dossier app.

Ne modifiez jamais AUCUN fichier dans le dossier `cake` pour éviter d'avoir des problèmes lors de futures mises à jour du coeur

3.5.3 Attributs des Contrôleurs

Les utilisateurs de PHP4 doivent inclure dans le code de leurs contrôleurs la définition de l'attribut $name. L'attribut $name sert à définir le nom du contrôleur. Comme indiqué précédemment il s'agit juste du nom du modèle, au pluriel. Cet attribut permet de contourner certaines limitations liées au nommage des classes en PHP4 et permet donc à CakePHP de résoudre les noms.

```
# exemple d'usage de l'attribut $name dans un contrôleur
<?php

class RecettesController extends AppController {
   var $name = 'Recettes';
}

?>
```

3.5.3.2 $components, $helpers et $uses

Les autres attributs les plus souvent utilisés permettent d'indiquer à CakePHP quels assistants (*helpers*), composants (*components*) et modèles vous utiliserez avec le contrôleur courant. Utiliser ces attributs rend ces classes MVC, fournies par `$components` et `$uses`, disponibles pour le contrôleur, sous la forme de variables de classe (`$this->NomModele`, par exemple) et celles fournies par `$helpers`, disponibles pour la vue

comme une variable référence à l'objet (`$nomassistant`).

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

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

Les assistants (*Helpers*) Html et Session sont toujours accessibles par défaut, tout comme le composant Session (*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 contrôleurs. Pour en savoir plus au sujet de ces classes, assurez-vous de regarder leurs sections respectives plus loin dans le manuel.

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

```
<?php

class RecettesController extends AppController {
    var $name = 'Recettes';
    var $uses = array('Recette', 'Utilisateur');
    var $helpers = array('Ajax');
    var $components = array('Email');
}

?>
```

Toutes ces variables sont fusionnées (merged) 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 contrôleur App.

Si vous ne souhaitez pas utiliser un modèle dans votre contrôleur, indiquez `var $uses = null` ou `var $uses = array()`. Ceci vous permettra d'utiliser un contrôleur sans la nécessité d'un fichier de modèle correspondant.

Il n'est pas conseillé d'ajouter tous les modèles systématiquement à votre contrôleur via le tableau `$uses`. Allez regarder ici et ici pour voir comment accéder proprement aux modèles respectivement associés ou pas.

3.5.3.3 Les attributs en relation avec la page : $layout

Quelques attributs sont à votre disposition dans les contrôleurs de CakePHP pour vous donner le contrôle sur la mise en page (*layout*) de vos vues.

L'attribut $layout peut ainsi prendre le nom de n'importe quel fichier de mise en page sauvegardé dans le répertoire /app/view/layout. Pour définir cet attribut il suffit d'y affecter le nom du fichier de mise en page moins son extension (.ctp). Si cet attribut n'est pas redéfini, CakePHP utilisera le fichier de mise en page par défaut, situé (ou à créer) dans /app/views/layout/default.ctp. Si vous n'avez pas redéfini ce fichier, CakePHP utilisera la mise en page par défaut (définie dans /cake/lib/view/layout/default.ctp).

```
# Utilisons $layout pour définir une mise en page alternative
<?php

class RecettesController extends AppController {
    function sauvegardeRapide() {
        $this->layout = 'ajax';
    }
}

?>
```

3.5.3.4 L'attribut Paramètres ($params)

Les paramètres d'un contrôleur sont accessible via `$this->params`. Cette variable est utilisée pour accéder aux informations relatives à la requête courante. La plupart du temps `$this->params` est utilisé pour accéder aux informations transmises au contrôleur via les opérations POST ou GET.

3.5.3.4.1 form

`$this->params['form']`

Toute donnée transmise par POST depuis tout formulaire est stockée dans cette variable, incluant également les informations trouvées dans $_FILES.

3.5.3.4.2 admin

`$this->params['admin']`

Il est défini à 1 si l'action courante a été invoquée via le routage d'admin.

3.5.3.4.3 bare

`$this->params['bare']`

Stocke 1 si le *layout* courant est vide, 0 s'il ne l'est pas.

3.5.3.4.4 isAjax

`$this->params['isAjax']`

Vaut 1 si la requête courante est un appel ajax, 0 sinon. Cette variable est seulement définie si le Composant RequestHandler est utilisé par le contrôleur.

3.5.3.4.5 controller

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

Stocke le nom du contrôleur courant qui manipule la requête. Par exemple, si l'URL **`/posts/voir/1`** était appelée, `$this->params['controller']` serait égal à "posts".

3.5.3.4.6 action

`$this->params['action']`

Stocke le nom de l'action courante exécutant la requête. Par exemple, si l'URL **`/posts/voir/1`** était interrogée, `$this->params['action']` serait égal à "voir".

3.5.3.4.7 pass

`$this->params['pass']`

Retourne un tableau (indexé numériquement) des paramètres d'URL situés après le nom de l'action.

```
// URL: /posts/voir/12/imprimer/reduire

Array
(
    [0] => 12
    [1] => imprimer
    [2] => reduire
)
```

3.5.3.4.8 url

`$this->params['url']`

Stocke l'URL courante interrogée, ainsi que les paires clé-valeur des variables passées en GET. Par exemple, si l'URL **`/posts/voir/?var1=3&var2=4`** était appelée, `$this->params['url']` devrait contenir :

```
[url] => Array
(
    [url] => posts/voir
    [var1] => 3
    [var2] => 4
)
```

3.5.3.4.9 data

`$this->data`

Utilisé pour traiter les données transmises au contrôleur, en POST, depuis les formulaires du Helper Form.

```
<?php

// Le Helper Form est utilisé pour créer un élément de
formulaire
$form->text('Utilisateur.prenom');
```

Lequel, une fois affiché, ressemble à quelque chose comme :

```
<input name="data[Utilisateur][prenom]" value="" type="text" />
```

Quand le formulaire est soumis au contrôleur via POST, les données sont visibles dans `this->data`.

```
// Le prénom renseigné peut se trouver ici :
$this->data['Utilisateur']['prenom'];
?>
```

3.5.3.4.10 prefix

`$this->params['prefix']`

Rempli par le préfixe de routage. Par exemple, cet attribut contiendrait la chaîne "admin" durant une requête à /admin/posts/uneaction.

3.5.3.4.11 named

`$this->params['named']`

Stocke tout paramètre nommé dans la chaîne de requête, sous la forme /clé:valeur/. Par exemple, si l'URL **`/posts/voir/var1:3/var2:4`** était demandée, `$this->params['named']` serait un tableau contenant :

```
[named] => Array
(
    [var1] => 3
    [var2] => 4
)
```

3.5.3.5 Autres attributs

L'attribut $cacheAction fournit de l'aide pour mettre en cache les vues, et l'attribut $paginate permet de définir les options qui seront utilisées par défaut pour la pagination. Pour plus d'informations sur ces deux attributs, jetez un oeil sur les sections qui leur sont dédiées dans ce manuel.

N'hésitez pas à faire un tour dans l'API pour voir le rôle des autres attributs de la classe contrôleur, il y a en effet plusieurs autres variables qui mériteraient également leur section dans ce manuel.

3.5.3.6 persistModel

Mettez-moi à jour !

Utilisé pour créer des instances, mises en cache, des modèles qu'un contrôleur utilise. Quand défini à *true*, tous les modèles reliés au contrôleur seront mis en cache. Ceci peut améliorer les performances dans de nombreux cas.

3.5.4 Méthodes des Contrôleurs

`set(string $variable, mixed $valeur)`

La méthode `set()` est la voie principale utilisée pour transmettre des données de votre contrôleur à votre vue. Une fois `set()` utilisée, la variable de votre contrôleur devient accessible par la vue.

```
<?php

// Dans un premier temps vous passez les données depuis le contrôleur :
$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.

Les clefs de votre tableau ne seront plus infléchies avant d'être assignées à la vue ('clef_avec_underscore' ne devient plus 'clefAvecUnderscore', etc.).

```
<?php

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

//rend $couleur, $type, and $prixDeBase
//disponibles dans la vue:

$this->set($data);

?>
```

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

```
<?php
$this->set('title_for_layout', 'Ceci est le titre de la page');
?>
```

render

`render(string $action, string $layout, string $fichier)`

La méthode `render()` est automatiquement appelée à la fin de chaque action exécutée par le contrôleur. 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 `rechercher()` de notre contrôleur RecettesController est demandée, le fichier de vue situé dans **`/app/view/recettes/rechercher.ctp`** sera utilisé.

```
class RecettesController extends AppController {
...
function rechercher() {
    // Affiche la vue située dans /views/recettes/rechercher.ctp
    $this->render();
}
...
}
```

Bien que CakePHP appellera cette fonction automatiquement à la fin de chaque action (à moins que vous n'ayez défini `$this->autoRender` à false), vous pouvez l'utiliser pour spécifier un fichier de vue alternatif en précisant le nom d'une action dans le contrôleur, via le paramètre `$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/views`. Cela permet un affichage direct des éléments, ce qui est très pratique lors d'appels ajax.

```
// Affiche l'élement situé dans /views/elements/ajaxretour.ctp
$this->render('/elements/ajaxretour');
```

Vous pouvez également spécifier un fichier alternatif en utilisant le troisième paramètre `$file`. Quant vous utilisez `$file`, n'oubliez pas d'utiliser les constantes globales de CakePHP (comme `VIEWS`).

Le second paramètre `$layout` vous permet de spécifier le fichier de mise en page qui sera utilisée pour afficher la vue.

Rendering a specific view

Si vous voulez afficher une autre vue de celle prévue par convention, vous pouvez le faire en appelant la méthode `render()` directement dans votre contrôleur. Une fois que vous avez appelée cette méthode, CakePHP chargera la vue que vous avez demandé d'afficher mais pas la vue prévue par convention.

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

Ce code chargera la vue `app/views/posts/autre_vue.ctp` au lieu de la vue `app/views/posts/mon_action.ctp`

3.5.4.2 Contrôle du flux

`redirect(string $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.

```
<?php
function reglerAchats() {
    // Placez ici la logique pour finaliser l'achat...
    if($succes) {
        $this->redirect(array('controller'=>'paiements',
'action'=>'remerciements'));
    } else {
        $this->redirect(array('controller'=>'paiements',
'action'=>'confirmation'));
    }
}
?>
```

Vous pouvez aussi utiliser une URL relative ou absolue avec le paramètre $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());
```

3.5.4.2.2 flash

`flash(string $message, string $url, integer $pause)`

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.

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

3.5.4.3 Méthodes de Callbacks

Les contrôleurs 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 contrôleur soient effectuées.

`beforeFilter()`

Cette fonction est exécutée avant chaque action du contrôleur. C'est un endroit pratique pour vérifier le statut d'une session ou les permissions d'un utilisateur.

`beforeRender()`

Cette méthode est appelée après l'action du contrôleur 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.

`afterFilter()`

Cette méthode est appelée après chaque action du contrôleur, et après que l'affichage soit terminé. C'est la dernière méthode du contrôleur qui est exécutée.

afterRender()

Appelée lorsque la vue correspondant à l'action a été affichée.

CakePHP supporte également des rappels (callbacks) liés au prototypage rapide (scaffolding).

_beforeScaffold($methode)

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

_afterScaffoldSave($methode)

$methode nom de la méthode appelée, soit edit soit update.

_afterScaffoldSaveError($methode)

$methode nom de la méthode appelée, soit edit soit update.

_scaffoldError($methode)

$methode nom de la méthode appelée exemple index, modifier, etc.

3.5.4.4 Autres méthodes utiles

Cette méthode charge en mémoire les modèles requis par le contrôleur. 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 contrôleurs 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.

3.5.4.4.2 referer

Retourne l'URL référente de la requête courante.

3.5.4.4.3 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 entêtes 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`

3.5.4.4.4 postConditions

postConditions(array $donnees, mixed $op, string $bool, boolean $exclusif)

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 modèle. 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 modèle Commande. Ensuite une action du

contrôleur peut utiliser les données postées par ce formulaire pour construire automatiquement les conditions de la recherche :

```
function index() {
    $conditions=$this->postConditions($this->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->data
array(
    'Commande' => array(
        'nb_items' => '4',
        'referrer' => 'Ye Olde'
    )
)
*/

//Trouvons les commandes qui ont au moins 4 items et qui contiennent 'Ye Olde'
$conditions = $this->postConditions(
    $this->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 utilisé 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.

3.5.4.4.5 paginate

`requestAction(string $url, array $options)`

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

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'));`

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

`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.

```
// controllers/commentaires_controller.php
class CommentairesController extends AppController {
    function derniers() {
        return $this->Commentaire->find('all', array('order' =>
'Commentaire.created DESC', 'limit' => 10));
    }
}
```

Si nous créons un élément simple pour appeler cette fonction :

```
// views/elements/derniers_commentaires.ctp

$commentaires = $this->requestAction('/commentaires/derniers');
foreach($commentaires as $commentaire) {
    echo $commentaire['Commentaire']['title'];
}
```

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

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

Ecrit de cette manière, dès que l'élément est affiché, une requête sera faite au contrôleur 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('derniers_commentaires', array('cache'=>'+1 hour'));
```

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` prends désormais des urls basées sur des tableau dans le style de cake :

```
echo $this->requestAction(array('controller' => 'articles', 'action' =>
'particuliers'), 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'.

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

Ceci en tant que tableau dans le requestAction serait alors :

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

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

A la différence d'autres endroits où les tableaux d'urls sont identiques aux chaînes d'url, requestAction les manipule différemment.

Lors de l'utilisation d'un tableau d'url en conjonction avec requestAction() vous devez spécifier **tous** les paramètres dont vous aurez besoin dans l'action demandée. Ceci inclut des paramètres comme `$this->data` et `$this->params['form']`. De plus, en passant tous les paramètres requis, *named* et *pass* doivent êtres conformes dans le second tableau, comme vu ci-dessus.

3.5.4.4.7 loadModel

`loadModel(string $modelClass, mixed $id)`

La fonction `loadModel` est très pratique quand vous avez besoin d'utiliser un modèle qui n'est pas le modèle par défaut du contrôleur ou ses modèles associés.

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

```
$this->loadModel('Utilisateur', 2);
$utilisateur = $this->Utilisateur->read();
```

3.6 Composants

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

CakePHP est également livré avec un fantastique ensemble de composants, que vous pouvez utiliser pour vous aider :

- Sécurité

- Sessions
- Listes de contrôle d'accès (*ACL*)
- Emails
- Cookies
- Authentification
- Traitement de requêtes

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

3.6.2 Configuration des Composants

De nombreux composants du coeur nécessitent une configuration. Quelques exemples : Auth, Cookie et Email. Toute configuration pour ces composants, et pour les composants en général, se fait dans la méthode `beforeFilter()` de vos contrôleurs.

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

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

Serait un exemple de configuration des variables de composants dans la méthode `beforeFilter()` de votre contrôleur.

Il est néanmoins possible qu'un composant ait besoin que certaines options de configuration soient définies dans la méthode `beforeFilter` du contrôleur qui l'exécute. A cette fin, certains composants autorisent la définition de certaines de leurs options dans le tableau `$components`

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

Consultez la documentation appropriée pour déterminer quelles options de configuration fournit chaque composant.

Les composants peuvent avoir les *callbacks* `beforeRender` et `beforeRedirect`, qui seront déclenchés respectivement avant que votre page ne soit rendue et avant une redirection.

Vous pouvez désactiver le déclenchement des *callbacks* en paramétrant la propriété `enabled` d'un composant à `false`.

3.6.3 Créer des Composants personnalisés

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 composant pour héberger cette logique partagée afin de l'utiliser dans plusieurs contrôleurs différents.

La première étape consiste à créer un nouveau fichier et une classe pour le composant. Créez le fichier dans /app/controllers/components/math.php. La structure de base pour le composant ressemblerait à quelque chose

comme ça :

```
<?php

class MathComponent extends Object {
    function faitUneOperationComplexe($montant1, $montant2) {
        return $montant1 + $montant2;
    }
}

?>
```

Prenez note que le composant Math étend Object et non Component. Etendre Component peut générer des problèmes de redirection infinie, lorsqu'il est combiné avec d'autres composants.

3.6.3.1 Inclure des Composants dans vos Contrôleurs

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

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

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

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

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

L'exemple ci-dessus passerait le tableau contenant "precision" et "generateurAleatoire" comme second paramètre, à la méthode `initialize()` du MathComponent.

Actuellement, cette syntaxe n'est implémentée par aucun des Composants du Coeur.

3.6.3.2 Classe d'accès MVC dans les Composants

Pour avoir accès à l'instance du contrôleur depuis votre composant nouvellement créé, vous devrez implémenter la méthode startup() ou initialize(). Cets méthodes spéciales reçoivent une référence vers le contrôleur comme premier paramètre et sont automatiquement appelées. La méthode initialize() est appelée

avant la méthode beforeFilter() du contrôleur, et la méthode startup() après beforeFilter(). Si pour une raison quelconque vous *ne* voulez *pas* que la méthode startup() soit appelée lorsque le contrôleur met tout en place, fixez la variable de classe $disableStartup à *true*.

Si vous voulez insérer un peu de logique avant qu'une méthode "beforeFilter()" du contrôleur n'ait été appelée, utilisez la méthode initialize() du composant.

```
<?php
class VerifieComponent extends Object {
    //appelée avant Controller::beforeFilter()
    function initialize(&$controller) {
        // sauvegarde la référence du contrôleur pour une utilisation
ultérieure
        $this->controller =& $controller;
    }

    //appelée après Controller::beforeFilter()
    function startup(&$controller) {
    }

    function redirigeAilleurs($valeur) {
        // utilise une méthode du contrôleur
        $this->controller->redirect($valeur);
    }
}
?>
```

Vous pourriez également vouloir utiliser d'autres composants dans un composant personnalisé. Pour ce faire, créez simplement une variable de classe $components (comme vous l'auriez fait dans un contrôleur) qui est un tableau contenant les noms des composants que vous souhaitez utiliser.

Seule la méthode `initialize` des sous-composants est appelée automatiquement.

```
<?php
class MonComponent extends Object {

    // Ce composant utilise d'autres composants
    var $components = array('Session', 'Math');

    function faitQuelquechose() {
        $resultat = $this->Math->faitUneOperationComplexe(1, 2);
        $this->Session->write('quelquechose', $resultat);
    }

}
?>
```

Accéder/utiliser un modèle dans un composant n'est généralement pas recommandé; cependant si après avoir évalué les différentes possibilités c'est ce que vous voulez faire, vous devrez instancier la classe de votre modèle et l'utiliser manuellement. Voici un exemple :

```
<?php
class MathComponent extends Object {
    function faitUneOperationComplexe($montant1, $montant2) {
        return $montant1 + $montant2;
```

```
    }

    function faitUneOperationSuperComplexe($montant1, $montant2) {
        $instanceUtilisateur = ClassRegistry::init('Utilisateur');
        $utilisateursTotaux = $instanceUtilisateur->find('count');
        return ($montant1 + $montant2) / $utilisateursTotaux;
    }
}
?>
```

3.6.3.3 Utilisez d'autres Composants dans votre Composant

Parfois, l'un de vos composants peut nécessiter l'usage d'un autre.

Vous pouvez inclure d'autres composants dans votre composant, exactement de la même manière que vous les incluez dans les contrôleurs : utilisez la propriété `$components`.

```
<?php
class CustomComponent extends Object {
    var $name = "Custom"; // le nom de votre composant
    var $components = array( "Existant" ); // l'autre composant que votre composant
utilise

    function initialize(&$controller) {
        $this->Existant->foo();
    }

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

```
<?php
class ExistantComponent extends Object {
    var $name = "Existant";

    function initialize(&$controller) {
        $this->Custom->bar();
    }

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

3.7 Modèles

Un Modèle est à la fois votre modèle de données et, en programmation orientée objet, 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 modèles, chacun étant associé avec un autre.

Voici un simple exemple de définition de modèle dans CakePHP :

```php
<?php

class Ingredient extends AppModel {
    var $name = 'Ingredient';
}

?>
```

Avec juste cette simple déclaration, le modèle 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 modèle Ingredient étend le modèle de l'application AppModel, lequel étend la classe Model interne de CakePHP. C'est cette classe Model du coeur qui fournit les fonctionnalités à l'intérieur de votre modèle Ingredient.

La classe intermédiaire AppModel est vide et si vous n'avez pas créé la vôtre, elle provient du répertoire **`/cake/`**. AppModel vous permet de définir des fonctionnalités qui doivent être rendues disponibles pour tous les modèles de votre application. Pour faire cela, vous avez besoin de créer votre propre fichier **`app_model.php`** qui se loge à la racine du **`/app/`**. A la création d'un projet en utilisant Bake ce fichier sera créer automatiquement pour vous.

Créez vos fichiers PHP de modèle dans le dossier **`/app/models/`** ou dans un sous-répertoire de **`/app/models`**. CakePHP le trouvera quelque soit sa place dans le dossier. Par convention le fichier doit avoir le même nom que la classe; dans cet exemple **`ingredient.php`**.

CakePHP créera dynamiquement un objet modèle pour vous si il ne peut pas trouver un fichier correspondant dans **`/app/models`**. Cela veut également dire que si votre fichier de modèle n'est pas nommé correctement (ex : **`Ingredient.php`** ou **`ingredients.php`**) CakePHP utilisera une instance de AppModel, plutôt que votre fichier de modèle "franc-tireur" (d'un point de vue CakePHP). Si vous essayez d'utiliser une méthode que vous avez définie dans votre modèle ou dans un comportement attaché à votre modèle 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 modèle et que vous devez, soit vérifier les noms de fichier, soit nettoyer les fichiers temporaires ou les deux.

Voyez aussi les Comportements (*Behaviors*), pour plus d'informations sur la façon d'appliquer une logique similaire à de multiples modèles.

La propriété `$name` est nécessaire en PHP4 mais optionnelle en PHP5.

Une fois votre modèle défini, il peut être accédé depuis vos Contrôleurs. CakePHP rend automatiquement un modèle disponible en accès, dès lors que son nom valide celui du contrôleur. Par exemple, un contrôleur nommé IngredientsController initialisera automatiquement le modèle Ingredient et y accédera par `$this->Ingredient`.

```php
<?php

class IngredientsController extends AppController {
    function index() {
        // Récupère tous les ingrédients et les transmet à la
vue :
        $ingredients = $this->Ingredient->find('all');
        $this->set('ingredients', $ingredients);
    }
}
```

```
?>
```

Les modèles associés sont accessibles à travers le modèle principal. Dans l'exemple suivant, Recette a une association avec le modèle Ingredient.

```
<?php
class RecettesController extends AppController {
    function index() {
     $ingredients = $this->Recette->Ingredient->find('all');
     $this->set('ingredients', $ingredients);
    }
}
?>
```

Si les modèles n'ont absolument AUCUNE association entre eux, vous pouvez utilisez Controller::loadModel() pour récupérer le modèle.

```
<?php
class RecettesController extends AppController {
    function index() {
       $recettes = $this->Recette->find('all');

       $this->loadModel('Voiture');
       $voitures = $this->Voiture->find('all');
       $this->set(compact('recettes', 'voitures'));
    }
}
?>
```

3.7.2 Créer les tables de la base de données

Bien que CakePHP puisse avoir des sources de données qui ne sont pas pilotées par une base de données, la plupart du temps, elles le sont. CakePHP est designer pour être agnostique et il travaillera avec MySQL, MSSQL, Oracle, PostgreSQL et d'autres. Vous pouvez créer vos tables de base de données telles que vous l'auriez fait normalement. Quand vous créez vos classes de Modèle, elles s'associeront automatiquement aux tables que vous avez créées.

Table : les noms sont par convention en minuscules et au pluriel, les noms de tables multi-mots séparés par des *underscores*. Par exemple, un Modèle de nom "Ingredient" s'attend à un nom de table "ingredients". Un Modèle de nom "InscriptionEvenement" s'attendrait à un nom de table "incription_evenements". CakePHP inspectera vos tables pour déterminer le type de données de chaque champ et utilisera cette information pour automatiser un certain nombre de fonctionnalité, tel que l'apparence des champs de formulaire dans une vue.

Les noms des champs sont par convention en minuscules et séparés par des *underscores*.

L'association du Modèle au nom de la table peut être contrôlée par l'utilisation de l'attribut `useTable`, expliqué plus loin dans ce chapitre.

Dans le reste de cette section, vous verrez comment CakePHP associe les types de champ de la base de données, à des types de donnée PHP et comment CakePHP peut automatiser des tâches suivant la manière

dont vos champs ont été définis.

3.7.2.1 Association des types de données par Base de données

Chaque RDMS (*Relational Database Management System*, en français SGBD : Système de gestion de bases de données relationnelles) définit les types de données de manières un peu différentes. Avec une classe "datasource" pour chaque système de base de données, CakePHP associe ces types à quelque chose qu'il reconnaît et crée une interface unifiée, peu importe le système de base de données sur lequel vous lancerez votre application.

Cette section décrit comment chacun d'eux est associé.

3.7.2.1.1 MySQL

Type CakePHP	Propriétés du champ
primary_key	NOT NULL auto_increment
string	varchar(255)
text	text
integer	int(11)
float	float
datetime	datetime
timestamp	datetime
time	time
date	date
binary	blob
boolean	tinyint(1)

Un champ *tinyint(1)* est considéré comme un booléen par CakePHP.

3.7.2.1.2 MySQLi

Type CakePHP	Propriétés du champ
primary_key	DEFAULT NULL auto_increment
string	varchar(255)
text	text
integer	int(11)
float	float
datetime	datetime
timestamp	datetime
time	time
date	date

binary	blob
boolean	tinyint(1)

3.7.2.1.3 ADOdb

Type CakePHP	Propriétés du champ
primary_key	R(11)
string	C(255)
text	X
integer	I(11)
float	N
datetime	T (Y-m-d H:i:s)
timestamp	T (Y-m-d H:i:s)
time	T (H:i:s)
date	T (Y-m-d)
binary	B
boolean	L(1)

3.7.2.1.4 DB2

Type CakePHP	Propriétés du champ
primary_key	not null generated by default as identity (start with 1, increment by 1)
string	varchar(255)
text	clob
integer	integer(10)
float	double
datetime	timestamp (Y-m-d-H.i.s)
timestamp	timestamp (Y-m-d-H.i.s)
time	time (H.i.s)
date	date (Y-m-d)
binary	blob
boolean	smallint(1)

3.7.2.1.5 Firebird/Interbase

Type CakePHP	Propriétés du champ
primary_key	IDENTITY (1, 1) NOT NULL
string	varchar(255)

text	BLOB SUB_TYPE 1 SEGMENT SIZE 100 CHARACTER SET NONE
integer	integer
float	float
datetime	timestamp (d.m.Y H:i:s)
timestamp	timestamp (d.m.Y H:i:s)
time	time (H:i:s)
date	date (d.m.Y)
binary	blob
boolean	smallint

3.7.2.1.6 MS SQL

Type CakePHP	Propriétés du champ
primary_key	IDENTITY (1, 1) NOT NULL
string	varchar(255)
text	text
integer	int
float	numeric
datetime	datetime (Y-m-d H:i:s)
timestamp	timestamp (Y-m-d H:i:s)
time	datetime (H:i:s)
date	datetime (Y-m-d)
binary	image
boolean	bit

3.7.2.1.7 Oracle

Type CakePHP	Propriétés du champ
primary_key	number NOT NULL
string	varchar2(255)
text	varchar2
integer	numeric
float	float
datetime	date (Y-m-d H:i:s)
timestamp	date (Y-m-d H:i:s)
time	date (H:i:s)
date	date (Y-m-d)

binary	bytea
boolean	boolean
number	numeric
inet	inet

3.7.2.1.8 PostgreSQL

Type CakePHP	Propriétés du champ
primary_key	serial NOT NULL
string	varchar(255)
text	text
integer	integer
float	float
datetime	timestamp (Y-m-d H:i:s)
timestamp	timestamp (Y-m-d H:i:s)
time	time (H:i:s)
date	date (Y-m-d)
binary	bytea
boolean	boolean
number	numeric
inet	inet

3.7.2.1.9 SQLite

Type CakePHP	Propriétés du champ
primary_key	integer primary key
string	varchar(255)
text	text
integer	integer
float	float
datetime	datetime (Y-m-d H:i:s)
timestamp	timestamp (Y-m-d H:i:s)
time	time (H:i:s)
date	date (Y-m-d)
binary	blob
boolean	boolean

3.7.2.1.10 Sybase

Type CakePHP	Propriétés du champ
primary_key	numeric(9,0) IDENTITY PRIMARY KEY
string	varchar(255)
text	text
integer	int(11)
float	float
datetime	datetime (Y-m-d H:i:s)
timestamp	timestamp (Y-m-d H:i:s)
time	datetime (H:i:s)
date	datetime (Y-m-d)
binary	image
boolean	bit

3.7.2.2 Titres

Un objet, au sens physique du terme, a souvent un nom ou un titre par le biais duquel on peut y faire référence. Une personne a un nom comme John ou Michel ou Gaston. Un billet de blog a un titre. Une catégorie a un nom.

En spécifiant un champ `title` ou `name`, CakePHP utilisera automatiquement cet intitulé dans plusieurs circonstances :

- Maquettage rapide (Scaffolding) - titres de page, étiquettes des balises fieldset
- Listes - normalement utilisé pour les menus déroulants `<select>`
- TreeBehavior - mise en ordre, vues arborescentes

Si vous avez un champ "title" *et* un champ "name" dans votre table, le champ "title" sera utilisé.

Si vous voulez utiliser autre chose que la convention, définissez `var $displayField = 'un_champ';`. Un seul champ peut être défini ici.

3.7.2.3 "created" et "modified" (ou "updated")

Ces deux champs sont automatiquement gérés lors des appels à la méthode save() du modèle CakePHP. A la création d'une nouvelle ligne, son champ `created` sera automatiquement rempli et son champ `modified` est mis à jour chaque fois que des changements sont faits. Notez qu'un champ nommé `updated` aura le même comportement que le champ `modified`.

Ces deux champs spéciaux doivent être de type DATETIME avec NULL comme valeur par défaut.

3.7.2.4 Utiliser les UUIDs comme Clés primaires

Les clés primaires sont normalement définies par un champ INT. La base de donnée autoincrémente le champ, en commençant par 1, pour chaque nouvel enregistrement ajouté. De façon alternative, si vous spécifiez votre clé primaire comme CHAR(36) ou BINARY(36), CakePHP génèrera automatiquement des UUIDs lorsque de nouveaux enregistrements sont créés.

Un UUID est une chaine de 32 bytes séparés par quatre tirets, pour un total de 36 caractères. Par exemple :

```
550e8400-e29b-41d4-a716-446655440000
```

Les UUIDs sont conçus pour être uniques, pas seulement au sein d'une même table, mais également entre les différentes tables et bases de données. Si vous avez besoin d'un champ qui reste unique quelque soit le système utilisé, alors les UUIDS sont une bonne approche.

3.7.3 Récupérer vos données

`find($type, $params)`

Find est, parmi toutes les fonctions de récupération de données des modèles, une véritable bête de somme multi-fonctionnelle. `$type` peut être `'all'`, `'first'`, `'count'`, `'list'`, `'neighbors'` ou `'threaded'`. Le type par défaut est `'first'`. Gardez à l'esprit que `$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.champ' => $cetteValeur), // tableau de conditions
    'recursive' => 1, // entier
    'fields' => array('Model.champ', 'Model.champ2'), // tableau de nom de champs
    'order' => 'Model.id', // chaîne ou tableau définissant le ORDER BY
    'group' => array('Model.champ'), // champs pour le GROUP BY
    'limit' => n, // entier
    'page' => n, // entier
        'offset'=> n, // entier
    'callbacks' => true //les autres valeurs possibles 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 comportements (*behaviors*) et, bien sÃ»r, dans vos propres méthodes de modèle.

Plus d'informations sur les *callbacks* des modèles sont disponibles ici

3.7.3.1.1 find('first')

`find('first', $params)`

'first' est type de recherche par défault et retournera un résultat, vous devriez utiliser ceci dans tous les cas où vous attendez un seul résultat. Ci-dessous, une paire d'exemples simples (code du contôleur) :

```
function une_fonction() {
   ...
```

```
    $this->Article->order = null; // redéfinition s'il l'est
déjà
    $articleADemiAleatoire = $this->Article->find();
    $this->Article->order = 'Article.created DESC'; // simule le fait que le
modèle ait un ordre de tri par défaut
    $dernierCree = $this->Article->find();
    $dernierCreeEgalement = $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
(
    [NomModele] => Array
        (
            [id] => 83
            [champ1] => valeur1,
            [champ2] => valeur2.
            [champ3] => valeur3
        )

    [NomModeleAssocie] => Array
        (
            [id] => 1
            [champ1] => valeur1,
            [champ2] => valeur2.
            [champ3] => valeur3
        )
)
```

Il n'y a aucun paramètre additionnel utilisé par `find('first')`.

3.7.3.1.2 find('count')

`find('count', $params)`

`find('count', $params)` retourne une valeur de type entier. Ci-dessous, une paire d'exemples simples (code du contôleur) :

```
function une_fonction() {
    ...
    $total = $this->Article->find('count');
    $en_attente = $this->Article->find('count', array('conditions' =>
array('Article.statut' => 'en attente')));
    $auteurs = $this->Article->Utilisateur->find('count');
    $auteursPublies = $this->Article->find('count', array(
        'fields' => 'DISTINCT Article.utilisateur_id',
        'conditions' => array('Article.statut !=' => 'en attente')
    ));
    ...
}
```

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).

Il n'y a aucun paramètre additionnel utilisé par `find('count')`.

3.7.3.1.3 find('all')

`find('all', $params)`

`find('all')` 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, une paire d'exemples simples (code du contôleur) :

```
function une_fonction() {
   ...
   $tousLesArticles = $this->Article->find('all');
   $en_attente = $this->Article->find('all', array('conditions' =>
array('Article.statut' => 'en attente')));
   $tousLesAuteurs = $this->Article->Utilisateur->find('all');
   $tousLesAuteursPublies = $this->Article->User->find('all', array('conditions'
=> array('Article.statut !=' => 'en attente')));
   ...
}
```

Dans l'exemple ci-dessus `$tousLesAuteurs` contiendra chaque utilisateur de la table utilisateurs, 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
        (
            [NomModele] => Array
                (
                    [id] => 83
                    [champ1] => valeur1
                    [champ2] => valeur2
                    [champ3] => valeur3
                )

            [NomModeleAssocie] => Array
                (
                    [id] => 1
                    [champ1] => valeur1
                    [champ2] => valeur2
                    [champ3] => valeur3
                )

        )
)
```

Il n'y a aucun paramètre additionnel utilisé par `find('all')`.

3.7.3.1.4 find('list')

`find('list', $params)`

`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 contôleur) :

```
function une_fonction() {
    ...
    $tousLesArticles = $this->Article->find('list');
    $en_attente = $this->Article->find('list', array('conditions' =>
array('Article.statut' => 'en attente')));
    $tousLesAuteurs = $this->Article->Utilisateur->find('list');
    $tousLesAuteursPublies = $this->Article->User->find('list', array('conditions'
=> array('Article.statut !=' => 'en attente')));
    ...
}
```

Dans l'exemple ci-dessus `$tousLesAuteurs` contiendra chaque utilisateur de la table utilisateurs, il n'y aura pas de condition appliquée à la recherche puisqu'aucune n'a été passée.

Les résultats d'un appel à `find('list')` seront 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 modèle est utilisé comme clé et le champ affiché (*display field* qui peut être configuré en utilisant l'attribut displayField du modèle) est utilisé pour la valeur. Quelques exemples complémentaires pour clarifier les choses :

```
function une_fonction() {
    ...
    $juste_les_pseudos = $this->Article->Utilisateur->find('list', array('fields'
=> array('Utilisateur.pseudo'));
    $correspondancePseudo = $this->Article->Utilisateur->find('list',
array('fields' => array('Utilisateur.pseudo', 'Utilisateur.prenom'));
    $groupesPseudo = $this->Article->Utilisateur->find('list', array('fields'
=> array('Utilisateur.pseudo', 'Utilisateur.prenom', 'Utilisateur.groupe'));
    ...
}
```

Avec l'exemple de code ci-dessus, les variables résultantes devraient ressembler à quelque chose comme çà :

```
 $juste_les_pseudos = Array
(
    //[id] => 'pseudo',
    [213] => 'AD7six',
    [25] => '_psychic_',
    [1] => 'PHPNut',
    [2] => 'gwoo',
    [400] => 'jperras',
)

$correspondancePseudo = Array
(
    //[pseudo] => 'prenom',
    ['AD7six'] => 'Andy',
    ['_psychic_'] => 'John',
    ['PHPNut'] => 'Larry',
    ['gwoo'] => 'Gwoo',
    ['jperras'] => 'JoÃ«l',
)

$usernameGroups = Array
(
    ['Utilisateur'] => Array
        (
            ['PHPNut'] => 'Larry',
            ['gwoo'] => 'Gwoo',
        )

    ['Admin'] => Array
        (
            ['_psychic_'] => 'John',
            ['AD7six'] => 'Andy',
            ['jperras'] => 'JoÃ«l',
        )

)
```

3.7.3.1.5 find('threaded')

`find('threaded', $params)`

`find('threaded', $params)` retourne un tableau imbriqué et est particulièrement approprié si vous voulez utiliser le champ `parent_id` des données de votre modèle, pour construire les résultats associés.
Ci-dessous, une paire d'exemples simples (code du contôleur) :

```
function une_fonction() {
   ...
   $toutesLesCategories = $this->Categorie->find('threaded');
   $uneCategorie = $this->Categorie->find('first', array('conditions' =>
array('parent_id' => 42)); // pas la racine
   $quelquesCategories = $this->Categorie->find('threaded', array(
    'conditions' => array(
        'Article.lft >=' => $uneCategorie['Categorie']['lft'],
        'Article.rght <=' => $uneCategorie['Categorie']['rght']
    )
   ));
   ...
}
```

Il n'est pas nécessaire d'utiliser le comportement Tree pour appliquer cette méthode - mais tous les résultats souhaités doivent être trouvables en une seule requête.

Dans l'exemple ci-dessus, `$toutesLesCategories` contiendra un tableau imbriqué représentant la structure entière de categorie. Le second exemple fait usage de la structure de données utilisée par le comportement Tree, qui retourne un résultat partiel, imbriqué pour `$uneCategorie` et tout ce qu'il y a sous elle. Les résultats d'un appel à `find('threaded')` seront de la forme suivante :

```
Array
(
    [0] => Array
        (
            [nomModele] => Array
                (
                    [id] => 83
                    [parent_id] => null
                    [champ1] => valeur1
                    [champ2] => valeur2
                    [champ3] => valeur3
                )

            [nomModeleAssocie] => Array
                (
                    [id] => 1
                    [champ1] => valeur1
                    [champ2] => valeur2
                    [champ3] => valeur3
                )
            [children] => Array
                (
                    [0] => Array
                        (
                            [nomModele] => Array
                                (
                                    [id] => 42
                                    [parent_id] => 83
                                    [champ1] => valeur1
                                    [champ2] => valeur2
                                    [champ3] => valeur3
                                )

                            [nomModeleAssocie] => 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' => 'nom 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.

Il n'y a aucun paramètre additionnel utilisé par `find('threaded')`.

3.7.3.1.6 find('neighbors')

`find('neighbors', $params)`

'neighbors' exécutera un find similaire à 'first', mais retournera la ligne précédant et suivant celle que vous requêtez. Ci-dessous, un exemple simple (code du contôleur) :

```
function une_fonction() {
   $voisins = $this->Article->find('neighbors', array('field' => 'id', 'value'
=> 3));
}
```

Vous pouvez voir dans cet exemple, les deux éléments requis par le tableau `$params` : field et value. Les autres éléments sont toujours autorisés, comme dans tout autre find (ex : si votre modèle agit comme un *containable*, alors vous pouvez spécifiez 'contain' dans `$params`). Le format retourné par un appel à `find('neighbors')` est de la forme :

```
Array
(
    [prev] => Array
        (
            [NomModele] => Array
                (
                    [id] => 2
                    [champ1] => valeur1
                    [champ2] => valeur2
                    ...
                )
            [NomModeleAssocie] => Array
                (
                    [id] => 151
                    [champ1] => valeur1
                    [champ2] => valeur2
                    ...
                )
        )
    [next] => Array
        (
            [NomModele] => Array
                (
                    [id] => 4
                    [champ1] => valeur1
                    [champ2] => valeur2
                    ...
                )
            [NomModeleAssocie] => Array
                (
                    [id] => 122
                    [champ1] => valeur1
                    [champ2] => valeur2
                    ...
                )
        )
)
```

Notez que le résultat contient toujours seulement deux éléments de premier niveau : prev et next.

3.7.3.2 findAllBy

```
findAllBy<fieldName>(string $value)
```

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.

Exemple de findAllBy<x> en PHP5	Fragment SQL correspondant
$this->Produit->findAllByEtatOrdre("3");	Produit.etat_ordre = 3
$this->Recette->findAllByType("Gâteau");	Recette.type = "Gâteau"
$this->Utilisateur->findAllByNomFamille("Anderson");	Utilisateur.nom_famille = "Anderson"
$this->Gateau->findById(7);	Cake.id = 7
$this->Utilisateur->findByNomUtilisateur("psychic");	Utilisateur.nom_utilisateur = "psychic"

Les utilisateurs de PHP4 doivent utiliser cette fonction un peu différemment, à cause d'une insensibilité à la casse en PHP4 :

Exemple de findAllBy<x> en PHP4	Fragment SQL correspondant
$this->Produit->findAllByEtat_ordre("3");	Produit.etat_ordre = 3
$this->Recette->findAllByType("Gâteau");	Recette.type = "Gâteau"
$this->Utilisateur->findAllByNom_famille("Anderson");	Utilisateur.nom_famille = "Anderson"
$this->Gateau->findById(7);	Cake.id = 7
$this->Utilisateur->findByNom_utilisateur("psychic");	Utilisateur.nom_utilisateur = "psychic"

Les fonctions findBy() fonctionnent comme find('first',...), tandis que les fonctions findAllBy() fonctionnent comme find('all',...).

Dans chaque cas, le résultat retourné est un tableau formaté exactement comme il le serait avec, respectivement, find() ou findAll().

3.7.3.3 findBy

```
findBy<nomChamp>(string $value)
```

Ces fonctions magiques peuvent être utilisées comme un raccourci pour rechercher dans vos tables selon 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.

Exemple findAllBy<x> en PHP5	Fragment SQL Correspondant
$this->Produit->findAllByEtatCommande('3');	Produit.etat_commande = 3

$this->Recette->findAllByType('Gâteau');	Recette.type = 'Gâteau'
$this->Utilisateur->findAllByNom('Dupont');	Utilisateur.nom = 'Dupont'
$this->Gateau->findById(7);	Gateau.id = 7
$this->Utilisateur->findByPseudo('psychic');	Utilisateur.pseudo = 'psychic'

Les utilisateurs de PHP4 doivent utiliser cette fonction un peu différemment à cause de l'insensibilité à la casse en PHP4 :

Exemple findAllBy<x> en PHP4	Fragment SQL correspondant
$this->Produit->findAllByEtat_commande('3');	Produit.etat_commande = 3
$this->Recette->findAllByType('Gâteau');	Recette.type = 'Gâteau'
$this->Utilisateur->findAllByNom_famille('Martin');	Utilisateur.nom_famille = 'Martin'
$this->Gateau->findById(7);	Gateau.id = 7
$this->Utilisateur->findByNom_usage('psychic');	Utilisateur.nom_usage = 'psychic'

Les fonctions findBy() fonctionnent comme find('first',...), alors que les fonctions findAllBy() fonctionnent comme find('all',...).

Dans les deux cas, le résultat retourné est un tableau formaté exactement comme il l'aurait été depuis un find() ou un findAll().

3.7.3.4 query

`query(string $query)`

Les appels SQL que vous ne pouvez pas ou ne voulez pas faire grâce aux autres méthodes de modèle (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 Sanitize de CakePHP, qui vous aide à nettoyer les données provenant des utilisateurs, des attaques par injection et *cross-site scripting*.

`query()` ne respecte pas $Model->cachequeries car cette fonctionnalité est par nature déconnectée de tout ce qui concerne l'appel du modèle. Pour éviter les appels au cache de requêtes, fournissez un second argument *false*, par exemple : `query($query, $cachequeries = false)`

`query()` utilise le nom de la table déclaré dans la requête comme clé du tableau de données retourné, plutôt que le nom du modèle. Par exemple,

```
$this->Photo->query("SELECT * FROM photos LIMIT 2;");
```

devrait retourner

```
Array
(
    [0] => Array
        (
            [photos] => Array
                (
                    [id] => 1304
                    [user_id] => 759
                )
        )

    [1] => Array
        (
            [photos] => Array
                (
                    [id] => 1305
                    [user_id] => 759
                )
        )
)
```

Pour utiliser le nom du modèle comme clé du tableau et obtenir un résultat cohérent avec ce qui est retournée par les méthodes *Find*, la requête doit être réécrite :

```
$this->Photo->query("SELECT * FROM photos AS Photo LIMIT 2;");
```

qui retourne

```
Array
(
    [0] => Array
        (
            [Photo] => Array
                (
                    [id] => 1304
                    [user_id] => 759
                )
        )

    [1] => Array
        (
            [Photo] => Array
                (
                    [id] => 1305
                    [user_id] => 759
                )
        )
)
```

Cette syntaxe et la structure de tableau correspondante sont valides pour MySQL seulement. Cake ne fournit aucune abstraction de données lorsqu'on exécute des requêtes manuellement, donc les résultats exacts pourront varier selon la base de données.

3.7.3.5 field

`field(string $nom, array $conditions = null, string $ordre = null)`

Retourne la valeur d'un unique champ, spécifié par `$nom`, du premier enregistrement correspondant aux $conditions ordonnées par $ordre. Si aucune condition n'est passée et que l'id du modèle 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`.

```
$modele->id = 22;
echo $modele->field('nom'); // affiche le nom de l'entrée d'id 22

echo $modele->field('nom', array('created <' => date('Y-m-d H:i:s')), 'created
DESC'); // affiche le nom de l'instance la plus récemment créée
```

3.7.3.6 read()

`read($fields, $id)`

`read()` est une méthode utilisée pour récupérer les données du modèle 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.

```
function beforeDelete($cascade) {
    ...
    $classement = $this->read('classement'); // récupère le classement de
l'enregistrement qui doit être supprimé.
    $nom = $this->read('name', $id2); // récupère le nom d'un second
enregistrement.
    $classement = $this->read('classement'); // récupère le classement de
ce second enregistrement.
    $this->id = $id3; //
    $this->Article->read(); // lit un troisième enregistrement.
    $enregistrement = $this->data // stocke le troisième enregistrement dans
$enregistrement
    ...
}
```


Notez que le troisième appel à `read()` retourne le classement du même enregistrement que celui lu avant. Cela est du au fait que `read()` modifie `Model::$id` pour toute valeur passée comme `$id`. Les lignes 6-8 démontrent comment `read()` modifie les données du modèle courant.

3.7.3.7 Conditions de recherche complexes

La plupart des appels de recherche de modèles 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.

Dans sa forme la plus simple, une requête basée sur un tableau ressemble à ceci :

```
$conditions = array("Billet.titre" => "Ceci est un billet");

//Exemple d'utilisation avec un modèle:
$this->Billet->find('first', array('conditions' => $conditions);
```

La structure ici est assez significative : Tous les billets dont le titre à pour valeur « Ceci est un billet » 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 modèle. 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 billets dont le titre n'est pas "Ceci est un billet" :

```
array("Billet.titre" => "<> Ceci est un billet")
```

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 billets dont le titre appartient à un ensemble de valeur données :

```
array(
    "Billet.titre" => array("Premier billet", "Second billet", "Troisième
billet")
)
```

Faire un NOT IN(...) correspond à trouver les billets dont le titre n'est pas dans le jeu de données passé :

```
array(
    "NOT" => array( "Billet.titre" => array("Premier billet", "Second billet",
"Troisième billet") )
)
```

Ajouter des filtres additionnels aux conditions est aussi simple que d'ajouter des paires clé/valeur au tableau :

```
array
(
    "Billet.titre" => array("Premier billet", "Second billet", "Troisième
```

```
billet"),
    "Billet.created >" => date('Y-m-d', strtotime("-2 weeks")
)
```

Vous pouvez également créer des recherche qui comparent deux champs de la base de données

```
array("Billet.created = Billet.modified")
```

L'exemple ci-dessus retournera les billets où la date de création est égale à la date de modification (ie les billets 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 :

```
array(
    'Modele.champ & 8 = 1',
    // autres conditions habituellement utilisées
)
```

Par défaut, CakePHP joint les conditions multiples avec l'opérateur booléen AND, ce qui signifie que le bout de code ci-dessus correspondra uniquement aux billets qui ont été créés durant les deux dernières semaines, et qui ont un titre correspondant à ceux donnés. Cependant, nous pouvons simplement trouver les billets qui correspondent à l'une ou l'autre des conditions :

```
array
("or" =>
    array
    (
        "Billet.titre" => array("Premier billet", "Second billet", "Troisième
billet"),
        "Billet.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 "NEST-ABLE". Admettons que vous ayez une relation hasMany/belongsTo entre Billets et Auteurs, ce qui reviendrait à un LEFT JOIN. Admettons aussi que vous vouliez trouver tous les billets 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 billets écrits par Bob :

```
array (
    "Auteur.nom" => "Bob",
    "or" => array
    (
        "Billet.titre LIKE" => "%magique%",
        "Billet.created >" => date('Y-m-d', strtotime("-2 weeks")
    )
)
```

Cake peut aussi vérifier les champs `null`. Dans cet exemple, la requête retournera les enregistrements où le titre du billet n'est pas `null` :

```
array ("not" => array (
        "Billet.titre" => null,
    )
)
```

Pour gérer les requêtes BETWEEN, vous pouvez utiliser ceci :

```
array('Billet.id 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 rapide exemple de réalisation d'une requête DISTINCT. Vous pouvez utiliser d'autres opérateurs, tels que MIN(), MAX(), etc., de la même manière

```
array('fields'=>array('DISTINCT (Utilisateur.nom) AS nom_de_ma_colonne'),
'order'=>array('Utilisateur.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.nom' => 'Le truc qui marche bien')
```

```
    ),
    '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 avons une table "utilisateurs" avec "id", "nom" et "statuts". Le statuts peut être "A", "B" ou "C". Et nous voulons récupérer tous les utilisateurs qui ont un statuts différent de "B" en utilisant une sous requête.

Pour pouvoir effectuer cela, nous allons appeler la source de données du modèle 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.

```
$conditionsSousRequete['"Utilisateur2"."status"'] = 'B';

$dbo = $this->Utilisateur->getDataSource();
$sousRequete = $dbo->buildStatement(
array(
'fields' => array('"Urilisateur2"."id"'),
'table' => $dbo->fullTableName($this->Utilisateur),
/> 'alias' => 'Utilisateur2',
'limit' => null,
'offset' => null,
'joins' => array(),
'conditions' =&gt; $conditionsSousRequete,
'order' => null,
'group' => null
),
$this->Utilisateur
);
$sousRequete = ' "Utilisateur"."id" NOT IN (' . $sousRequete . ') ';
```

```
$expressionSousRequete = $dbo->expression($sousRequete);

$conditions[] = $expressionSousRequete;

$this->Utilisateur->find('all', compact('conditions'));
```

Ceci devrait généré la commande SQL suivante :

```
SELECT
"Utilisateur"."id" AS "Utilisateur__id",
"Utilisateur"."nom" AS "Utilisateur__nom",
"Utilisateur"."status" AS Utilisateur__status"
FROM
"utilisateurs" AS "Utilisateur"
WHERE
"Utilisateur"."id" NOT IN (
SELECT
"Utilisateur2"."id"
FROM
"utilisateurs" AS "Utilisateur2"
WHERE
"Utilisateur2"."status" = 'B'
)
```

3.7.4 Sauvegarder vos données

CakePHP rend la sauvegarde des données d'un modèle très rapide. Les données prêtes à être sauvegardées doivent être passées à la méthode `save()` du modèle 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 `HtmlHelper`, `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->data` pour un usage rapide et pratique.

Voici un exemple simple d'une action de contrôleur qui utilise un modèle CakePHP pour sauvegarder les données dans une table de la base de données :

```
function modifier($id) {
    //Est-ce que des données de formulaires ont été POSTées ?

    if(!empty($this->data)) {
        //Si les données du formulaire peuvent être validées et
sauvegardées ...

        if($this->Recette->save($this->data)) {
```

```
            //On définit une message flash en session et on redirige.

            $this->Session->setFlash("Recette sauvegardée !");
            $this->redirect('/recettes');
        }
    }

    //Si aucune données de formulaire, on récupère la recette
à éditer
    //et on la passe à la vue


    $this->set('recette', $this->Recette->findById($id));
}
```

Note additionnelle : 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 Données 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.

Il y a quelques autres méthodes du modèle liées à la sauvegarde que vous trouverez utiles :

`save(array $donnees = null, boolean $valider = true, array $listeDesChamps = array())`

Model::set() peut être utilisée pour définir un ou plusieurs champs de données dans le tableau des données d'un modèle. Ceci est utile quand on utilise des modèles en conjonction avec ActiveRecord

```
$this->Article->read(null, 1);
$this-&gt;Article->set('title', 'Nouveau titre pour cet article');
$this->Article->save();
```

Est un exemple de comment vous pouvez utiliser `set()` pour mettre à jour et sauvegarder un champ simple, dans une approche "ActiveRecord". Vous pouvez aussi utiliser `set()` pour assigner de nouvelles valeurs à plusieurs champs.

```
$this-&gt;Article->read(null, 1);
$this->Article->set(array(
/> 'title' => 'Nouveau titre',
'publie' => false
));
$this->Article->save();
```

Le code si dessus mettra à jour le titre et le champ "publié" et les sauvegardera dans la base de données.

`save(array $donnees = null, boolean $valide = true, array $listeDeChamps = 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 modèle devant être sauvegardés. Pour une sécurité accrue, vous pouvez limiter les champs sauvegardés à ceux listés dans `$listeDesChamps`.

Si `$listeDeChamps` n'est pas fourni, un utilisateur malicieux peut ajouter des champs additionnels dans le formulaire de données, et ainsi changer la valeur de champs qui n'étaient pas prévus à l'origine.

La méthode `save()` a également une syntaxe alternative :

`save(array $donnees = null, array $parametres = array())`

Le tableau `$parametres` peut avoir n'importe laquelle des options suivante comme clés :

```
array(
    'validate' => true,
    'fieldList' => array(),
    'callbacks' => true //o autres valeurs possibles : false, 'before', 'after'
)
```

Plus d'informations sur les callbacks de modèle est disponible ici

Une fois qu'une sauvegarde a été effectuée, l'ID de l'objet peut-être trouvé dans l'attribut `$id` de l'objet modèle - ceci est particulièrement pratique lorsque l'on crée de nouveaux objets.

```
$this->Ingredient->save($nouvelleDonnees);
$idDuNouvelIngredient = $this->Ingredient->id;
```

La création ou la mise à jour est contrôlée par le champ `id` du modèle. Si `$modele->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->Recette->create();
$this->Recette->save($this->data);

//Mise à jour: id est défini à une valeur numérique
$this->Recette->id = 2;
$this->Recette->save($this->data);
```

Lors de l'appel à `save()` dans une boucle, n'oubliez pas d'appeler `create()`.

`create(array $donnees = array())`

Cette méthode initialise la classe du modèle pour sauvegarder de nouvelles informations.

Si vous renseignez le paramètre `$data` (en utilisant le format de tableau mentionné plus haut), le nouveau modèle créé sera prêt à être sauvegardé avec ces données (accessibles à `$this->data`).

Si `false` est passé à la place d'un tableau, l'instance du modèle n'initialisera pas les champs du schéma de modèle qui ne sont pas encore définis, cela remettra à zéro les champs qui ont déjà été renseignés, et laissera les autres vides. Utilisez ceci pour éviter de mettre à jour des champs de la base données qui ont déjà été renseignés et doivent être mis à jour.

`saveField(string $nomDuChamp, string $valeurDuChamp, $valider = false)`

Utilisé pour sauvegarder la valeur d'un seul champ. Fixez l'ID du modèle (`$this->NomDuModele->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 modèle et du champ.

Par exemple, pour mettre à jour le titre d'un article de blog, l'appel depuis un contrôleur à `saveField` ressemblerait à quelque chose comme :

```
$this->Post->saveField('title', 'Un nouveau titre pour une nouvelle
journée');
```

updateAll(array $champs, array $conditions)

Met à jour plusieurs enregistrements en un seul appel. Les enregistrements à mettre à jour sont identifiés par le tableau `$conditions`, et les champs devant être mis à jour, ainsi que leurs valeurs, sont identifiés par le tableau `$champs`.

Par exemple, si je voulais approuver tous les utilisateurs qui sont membres depuis plus d'un an, l'appel à update devrait ressembler à quelque chose du style :

```
$cette_annee = date('Y-m-d h:i:s', strtotime('-1 year'));

$this->Utilisateur->updateAll(
    array('Utilisateur.created' => "<= $cette_annee"),
    array('Utilisateur.approuve' => true)


);
```

Le tableau `$champs` accepte des expressions SQL. Les valeurs litérales doivent être manuellement quotées.

Par exemple, pour fermer tous les tickets appartenant à un certain client :

```
$this->Ticket->updateAll(
    array('Ticket.statut' => "'fermé'"),
    array('Ticket.client_id' => 453)
);
```

saveAll(array $donnees = null, array $options = array())

Utilisé pour pour sauvegarder (a) des enregistrements individuels multiples pour un seul modèle ou (b) une entrée, et tous les enregistrements associés

Les options suivantes peuvent être utilisées :

validate : mettre à `false` pour désactiver la validation, à `true` pour valider chaque enregistrement avant sauvegarde, "first" pour valider *tous* les enregstrements avant que l'un d'entre eux soit sauvegardé, ou "only" pour simplement valider les enregistrements sans les sauvegarder.

atomic : si `true` (valeur par défaut), cela tentera de sauvegarder tous les enregistrements dans une même transaction. Doit être fixé à `false` si la base de données/table ne supporte pas les transactions. Si `false`, on renvoie un tableau similaire au tableau $donnees passé, mais les valeurs sont mises à true/false selon si chaque enregistrement a été sauvegardé avec succès ou non.

fieldList: équivalent au paramètre $fieldList de `Model::save()`

Pour sauvegarder des enregistrements multiples d'un même modèle, $donnees doit être un tableau indexé numériquement comme ceci :

```
Array
(
        [0] => Array
                (
                        [title] => titre 1
                )
        [1] => Array
                (
                        [title] => titre 2
                )
)
```

La commande pour sauvegarder le tableau $donnees ci-dessus serait :

```
$this->Article->saveAll($donnees['Article']);
```

Pour sauvegarder un enregistrement et tous ces enregistrements liés par une association hasOne ou belongsTo, le tableau de données doit ressembler à :

```
Array
(
    [Utilisateur] => Array
        (
            [pseudo] => billy
        )
    [Profil] => Array
        (
            [sexe] => Homme
                        [emploi] => Programmeur
        )
)
```

La commande pour sauvegarder le tableau $donnees ci-dessus serait :

```
$this->Article->saveAll($donnees);
```

Pour sauvegarder un enregistrement et tous ces enregistrements liés par une association hasMany, le tableau de données doit ressembler à :

```
Array
(
    [Article] => Array
        (
            [title] => Mon premier article
        )
    [Commentaire] => Array
        (
            [0] => Array
                (
                    [commentaire] => Commentaire 1
                    [utilisateur_id] => 1
```

```
                )
            [1] => Array
                (
                    [commentaire] => Commentaire 2
                    [utilisateur_id] => 2
                )
        )
)
```

La commande pour sauvegarder le tableau $donnees ci-dessus serait :

```
$this->Article->saveAll($donnees);
```

Sauvegarder des données en relation à l'aide de `saveAll()` ne marchera que pour les modèles qui ont effectivement une (ou plusieurs) relation définie.

3.7.4.1 Sauvegarder les données des modèles liés (hasOne, hasMany, belongsTo)

Quand on travaille avec des modèles associés, il est important de réaliser que sauvegarder les données d'un modèle doit toujours se faire depuis le modèle CakePHP correspondant. Si vous sauvegardez un nouveau Post et ses Commentaires associés, vous devrez alors utiliser à la fois les modèles Post et Commentaire durant l'opération de sauvegarde.

Si aucun enregistrement des modèles associés n'existe dans le système à ce moment là (par exemple si vous souhaitez sauvegarder un nouvel Utilisateur et ses enregistrements de Profil liés en même temps), vous devrez d'abord sauvegarder le modèle primaire (ou parent).

Pour avoir une idée de comment cela fonctionne, imaginons que nous ayons une action dans notre contrôleur UtilisateurController qui permette de sauvegarder un nouvel Utilisateur et un Profil lié. L'exemple d'action ci-dessous supposera que vous ayez POSTé suffisamment de données (en utilisant le FormHelper) pour créer un Utilisateur et un Profil.

```
<?php
function add() {
    if (!empty($this->data)) {


        // On peut sauvegarder les données Utilisateur
        // elles devraient être dans $this->data['Utilisateur']

        $utilisateur = $this->Utilisateur->save($this->data);



        // Si l'utilisateur a été sauvegardé nous ajoutons cette
information aux données à sauvegarder
        // et sauvegardons le Profil


        if (!empty($utilisateur)) {
            // L'ID de l'Utilisateur nouvellement créé a été
stockée dans
            // $this->Utilisateur->id.
            $this->data['Profil']['utilisateur_id'] = $this->Utilisateur->id;
```

```
            // Car Utilisateur hasOne Profil, nous pouvons accéder
            // au modèle Profil à travers le modèle Utilisateur :
            $this->Utilisateur->Profil->save($this->data);
        }
    }
}
?>
```

Une règle lorsque l'on travaille avec les associations hasOne, hasMany et belongsTo : tout n'est que manipulation de clés. L'idée de base est de prendre la clé d'un modèle et de la placer dans le champ de clé étrangère de l'autre. Quelquefois, cela implique l'utilisation de l'attribut `$id` de la classe de modèle après une sauvegarde (`save()`), mais dans d'autres cas il s'agit simplement de la récupération d'un ID depuis le champ caché du formulaire qui a été POSTé vers une action de contrôleur.

En complément de l'approche basique utilisée ci-dessus, CakePHP offre une méthode `saveAll()` très pratique, qui permet de valider et de sauvegarder des modèles multiples en une seule fois. De plus, `saveAll()` fournit un support transactionnel pour assurer l'intégrité des données dans votre base de données (càd que si votre modèle ne parvient pas à se sauvegarder, les autres modèles ne seront pas sauvegardés non plus).

Pour que les transactions fonctionnent correctement dans MySQL vos tables doivent utiliser InnoDB. Rappelez-vous que le tables MyISAM ne supportent pas les transactions.

Regardons comment nous pouvons utiliser `saveAll()` pour sauvegarder les modèles Entreprise et Compte en même temps.

D'abord, vous devrez construire votre formulaire à la fois pour les modèles Entreprise et Compte (nous supposerons qu'une Entreprise *hasMany* Compte).

```
echo $form->create('Entreprise', array('action'=>'ajouter'));
echo $form->input('Entreprise.nom', array('label'=>'Nom de l\'entreprise'));
echo $form->input('Entreprise.description');
echo $form->input('Entreprise.localisation');

echo $form->input('Compte.0.nom', array('label'=>'Nom du compte'));
echo $form->input('Compte.0.login');
echo $form->input('Compte.0.email');

echo $form->end('Ajouter');
```

Jetons un oeil à la manière dont nous avons nommé les champs du formulaire pour le modèle Compte. Si Entreprise est notre modèle principal, `saveAll()` s'attendra à ce que les données des modèles liés (Compte) arrivent dans un format spécifique. Ainsi, `Compte.0.nomDuChamp` est exactement ce dont nous avons besoin.

Le nommage des champs ci-dessus est nécessaire pour une association *hasMany*. Si l'association entre les modèles est de type *hasOne*, il faudra utiliser la notation NomDuModele.nomChamp pour le modèle associé.

Maintenant, dans notre contrôleur entreprises_controller nous pouvons créer une action `ajouter()` :

```
function ajouter() {
   if(!empty($this->data)) {
      $this->Entreprise->saveAll($this->data, array('validate'=>'first'));
   }
```

```
}
```

C'est tout ce qu'il y a à faire. Désormais nos modèles Entreprise et Compte seront tous deux validés et sauvegardés au même moment. Une chose à noter, est l'utilisation ici de `array('validate'=>'first')`, cette option nous assure que les deux modèles seront validés.

3.7.4.1.1 counterCache - Mettez en cache vos count()

Cette fonction vous aide à mettre en cache le décompte des données associées. Plutôt que de compter les enregistrements manuellement, avec find('count'), le modèle traque lui-même, tout ajout/suppression dans le modèle associé par $hasMany et incrémente/décrémente un champ integer dédié, dans la table du modèle parent.

Le nom de ce champ est constitué du nom du modèle au singulier suivi par un tiret bas (*underscore*) et du mot `count`.

```
mon_modele_count
```

Imaginons que vous ayez un modèle appelé `ImageCommentaire` et un modèle appelé `Image`, vous ajouteriez un nouveau champ INT à la table `image` et le nommeriez `image_commentaire_count`.

Voici d'autres exemples:

Modèle	Modèle associé	Example
Utilisateur	Image	utilisateurs.image_count
Image	ImageCommentaire	images.images_commentaires_count
BlogArticle	BlogArticleCommentaire	blog_articles.blog_article_commentaire_count

Dès que vous avez ajouter un champ compteur, tout est bon. Activer le counter-cache dans votre association en ajoutant une clef `counterCache` et en paramétrant la valeur sur `true`.

```
class Image extends AppModel {
    var $belongsTo = array(
        'ImageAlbum' => array('counterCache' => true)
    );
}
```

Maintenant, chaque fois que vous ajoutez ou retirez une nouvelle `Image` à `ImageAlbum`, le nombre dans `image_count` sera ajusté en conséquence.

Vous pouvez aussi spécifier un `counterScope`. Il vous permet essentiellement de spécifier une condition simple, qui indique au modèle quand se mettre à jour (ou pas, cela dépend de votre façon de voir les choses) la valeur du compteur.

En reprenant l'exemple de notre modèle Image, nous pouvons le spécifier ainsi :

```
class Image extends AppModel {
```

```
    var $belongsTo = array(
        'ImageAlbum' => array(
            'counterCache' => true,
            'counterScope' => array('Image.active' => 1) // Compte seulement si
'Image est actif = 1
    ));
}
```

3.7.4.2 Sauvegarder les données des modèles liés (HABTM)

Sauvegarder des modèles qui sont associés par un hasOne, belongsTo et hasMany est relativement simple : vous n'avez qu'à renseigner la clé étrangère avec l'ID du modèle associé. Une fois que ceci est fait, il vous suffit d'appeler la méthode save() sur le modèle et tout se reliera correctement.

Avec les relations HABTM, vous devez fixer l'ID du modèle associé dans votre tableau de données. Nous allons construire un formulaire qui crée un nouveau tag et l'associe à la volée à une recette.

Le formulaire le plus simpliste ressemblerait à quelque chose comme ceci (nous supposerons que $recette_id contient déjà une valeur) :

```
<?php echo $form->create('Tag');?>
    <?php echo $form->input(
        'Recette.id',
        array('type'=>'hidden', 'value' => $recette_id)); ?>
    <?php echo $form->input('Tag.nom'); ?>
    <?php echo $form->end('Ajouter le tag'); ?>
```

Dans cet exemple, vous pouvez remarquer le champ caché `Recette.id` dont la valeur est l'ID de la recette à laquelle nous voulons relier le tag.

Quand la méthode `save()` est invoquée dans le contrôleur, elle sauvera automatiquement les données HABTM dans la base de données.

```
function add() {
    // Sauvegarde l'association
    if ($this->Tag->save($this->data)) {
        // Faire quelque chose en cas d'enregistrement réussi
    }
}
```

Avec le code précédent, notre nouveau Tag est créé et associé avec une Recette, dont l'ID était défini dans $this->data['Recette']['id'].

Pour d'autres raisons nous pourrions vouloir présenter les données associées sous forme de liste déroulante. Les données peuvent être récupérées depuis le modèle en utilisant la méthode `find('list')` et assignées à une variable de vue au nom du modèle. Un champ input avec le même nom sera automatiquement affiché comme un `select` contenant les données.

```
// dans le contrôleur
$this->set('tags', $this->Recette->Tag->find('list'));

// dans la vue
$form->input('tags');
```

Un scénario plus probable avec une relation HABTM inclurait un ensemble de `select` pour permettre des sélections multiples. Par exemple, une Recette peut avoir de multiples Tags qui lui sont assignés. Dans ce cas, les données sont tirées du modèle de la même manière, mais le champ de formulaire est déclaré de manière un peu différente. Le nom du tag est défini en utilisant la convention `NomModele`.

```
// dans le contrôleur
$this->set('tags', $this->Recette->Tag->find('list'));

// dans la vue
$form->input('Tag');
```

En utilisant le code précédent, un menu déroulant de sélection multiple est créé, permettant la sauvegarde automatique de choix multiples pour la Recette existante, lors d'un ajout ou d'une sauvegarde dans la base de données.

Que faire quand la relation HABTM devient compliquée ?

Par défaut, en sauvant une relation "HasAndBelongsToMany", Cake effacera toutes les lignes de la table de jointure avant de sauvegarder les nouvelles. Par exemple, si vous avez un "Club" qui a 10 "Affilié" associés, et que vous mettez à jour le "Club" avec 2 nouveaux "Affilié", le "Club" ne comptera que 2 "Affilié", et pas 12.

Notez également que, si vous voulez ajouter d'autres champs à la table de jointure (quand il a été créé ou des métas informations), c'est possible avec les tables de jointure HABTM, mais il est important de comprendre que vous avez une option plus facile.

Une relation HABTM entre deux modèles est en réalité un raccourci pour trois modèles associés deux à deux par des relations hasMany et belongsTo.

Considérons cet exemple :

```
Affilié hasAndBelongsToMany Club
```

Une autre façon de voir les choses est d'ajouter un modèle "Abonnement"

```
Affilié hasMany Abonnement
Abonnement belongsTo Affilié, Club
Club hasMany Abonnement
```

Ces deux exemples sont quasiment identiques. Ils utilisent le même nombre de champs dans la base de données et le même nombre de modèles. Les différences importantes sont que le modèle de jointure est nommé différemment et que son comportement est plus prévisible.

Quand votre table de jointure contient des champs en plus des clés étrangères, la plupart du temps, il est plus facile de créer un modèle pour la jointure et des relations "hasMany" et "belongsTo" comme dans l'exemple ci-dessus, au lieu d'utiliser des relations HABTM.

3.7.5 Supprimer des données

```
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 Utilisateur lié à plusieurs enregistrements Recette :

- si $cascade est fixé à *true*, les entrées Recette liées sont aussi supprimées si les valeurs "dependant" des modèles sont à *true*.
- si $cascade est fixé à *false*, les entrées Recette resteront après que l'Utilisateur ait été supprimé.

3.7.5.2 deleteAll

```
deleteAll(mixed $conditions, $cascade = true, $callbacks = false)
```

Identique à `delete()` et `remove()`, sauf que `deleteAll()` supprime tous les enregistrements correspondant aux conditions fournies. Le tableau `$conditions` doit être un fragment SQL ou un tableau.

3.7.6 Associations : relier les modèles entre eux

Une des caractéristiques les plus puissantes de CakePHP est sa capacité d'établir les liens nécessaires entre les modèles d'après les informations fournies. Dans CakePHP, les liens entre modèles sont gérés par des associations.

Definir 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 modèles 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 modèles de Plugin, voyez Modèles de plugins.

3.7.6.1 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 utilisateur a un profil.
un vers plusieurs	hasMany	Un utilisateur peut avoir plusieurs recettes.
plusieurs vers un	belongsTo	Plusieurs recettes appartiennent à un utilisateur.
plusieurs vers plusieurs	hasAndBelongsToMany	Les recettes ont, et appartiennent à plusieurs tags.

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.

```
<?php

class Utilisateur extends AppModel {
    var $name = 'Utilisateur ';
    var $hasOne = 'Profil';
    var $hasMany = array(
        'Recette' => Array
            'className'  => 'Recette',
            'conditions' => array('Recette.acceptee' => '1'),
            'order'      => 'Recette.created DESC'
        )
    );
}

?>
```

Dans l'exemple ci-dessus, la première instance du mot 'Recette' 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 doivent être uniques à la fois dans un modèle et de part et d'autre d'une relation belongsTo/hasMany ou belongsTo/hasOne. Choisir des noms non-uniques pour des alias de modèle peut engendrer des comportements non souhaités.

Cake créera automatiquement des liens entre les objets de modèles associés. Par exemple dans notre modèle `Utilisateur` vous pourrez accéder au modèle `Recette` par

```
$this->Recette->uneFonction();
```

De la même manière dans votre contrôleur vous pouvez accéder à un modèle associé en suivant simplement les associations de modèle et sans l'ajouter dans le tableau `$uses` :

```
$this->Utilisateur->Recette->uneFonction();
```

Rappelez-vous que les associations sont définies 'dans un seul sens'. Si vous définissez Utilisateur *hasMany* Recette cela n'aura aucun effet sur le modèle Recette. Vous devrez définir Recette *belongsTo* Utilisateur pour pouvoir accéder au modèle Utilisateur depuis le modèle Recette.

3.7.6.2 hasOne

Mettons en place un modèle Utilisateur avec une relation de type hasOne vers un modèle 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 utilisateur_id. Le motif de base est :

hasOne: l'*autre* modèle contient la clé étrangère.

Relation	Schéma

Pomme hasOne Banane	bananes.pomme_id
Utilisateur hasOne Profil	profils.utilisateur_id
Docteur hasOne Maitre	maitres.docteur_id

Le fichier du modèle Utilisateur sera sauvegardé dans /app/models/utilisateur.php. Pour définir l'association 'Utilisateur hasOne Profil', ajoutez la propriété $hasOne à la classe du modèle. Pensez à avoir un modèle Profil dans /app/models/profil.php, sans quoi l'association ne fonctionnera pas.

```
<?php

class Utilisateur  extends AppModel {
    var $name = 'Utilisateur ';
    var $hasOne = 'Profil';
}
?>
```

Il y a deux manières de décrire cette relation dans vos fichiers de modèle. La méthode la plus simple est de fixer comme valeur à l'attribut $hasOne une chaîne de caractères contenant le nom de la classe associée au modèle, comme nous l'avons fait ci-dessus.

Si vous avez besoin de plus de contrôle, vous pouvez définir vos associations en utilisant un tableau. Par exemple, vous pourriez vouloir classer les colonnes par date décroissante, ou limiter l'association afin qu'elle n'inclue que certains enregistrements.

```
<?php

class Utilisateur  extends AppModel {
    var $name = 'Utilisateur ';
    var $hasOne = array(
        'Profile' => array(
            'className'    => 'Profil',
            'conditions'   => array('Profil.publie' => '1'),
            'dependent'    => true
        )
    );
}
?>
```

Les clés possibles pour un tableau décrivant une association $hasOne sont :

- **className** : le nom de la classe du modèle que l'on souhaite associer au modèle actuel. Si l'on souhaite définir la relation 'Utilisateur a un Profil', la valeur associée à la clé 'className' devra être 'Profil'.
- **foreignKey** : le nom de la clé etrangère que l'on trouve dans l'autre modèle. 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 modèle actuel (avec des underscores) suffixé avec '_id'. Dans l'exemple ci-dessus la valeur par défaut aurait été 'utilisateur_id'.
- **conditions** : un fragment de code SQL utilisé pour filtrer les enregistrements du modèle relié. C'est une bonne pratique que d'utiliser les noms des modèles dans ces portions de code : "Profil.approuve = 1" sera toujours mieux qu'un simple "approuve = 1".
- **fields** : une liste des champs à récupérer lorsque les données du modèle associé sont parcourues. Par défaut, cela retourne tous les champs.

- **order** : un fragment SQL qui définit l'ordre pour les lignes de résultats retournées.
- **dependent** : lorsque la valeur de la clé 'dependent' est true et que la méthode delete() du modèle est appelée avec le paramètre 'cascade' valant true également, les enregistrements des modèles associés sont supprimés. Dans ce cas nous avons fixé la valeur à true de manière à ce que la suppression d'un Utilisateur supprime également le Profil associé.

Une fois que cette association aura été définie, les opérations de recherche sur le modèle Utilisateur récupèreront également les enregistrements Profils liés s'il en existe :

```
//Exemple de résultats d'un appel à $this->Utilisateur->find().

Array
(
    [Utilisateur] => Array
        (
            [id] => 121
            [nom] => Gwoo le Kungwoo
            [created] => 2007-05-01 10:31:01
        )
    [Profil] => Array
        (
            [id] => 12
            [utilisateur_id] => 121
            [competences] => Cuisiner des gâteaux
            [created] => 2007-05-01 10:31:01
        )
)
```

3.7.6.3 belongsTo

Maintenant que nous avons accès aux données du Profil depuis le modèle Utilisateur, définissons une association belongsTo (appartient a) dans le modèle Profil afin de pouvoir accéder aux données Utilisateur 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 modèle *actuel* contient la clef étrangère.

Relation	Schéma
Banane belongsTo Pomme	bananes.pomme_id
Profil belongsTo Utilisateur	profils.utilisateur_id
Maitre belongsTo Docteur	maitres.docteur_id

Si un modèle (table) contient une clé étrangère, elle appartient à (*belongsTo*) l'autre modèle (table).

On peut définir l'association belongsTo dans notre modèle Profil (/app/models/profil.php) en utilisant une chaîne de caractère de cette manière :

```
<?php
```

```
class Profil extends AppModel {
    var $name = 'Profil';
    var $belongsTo = 'Utilisateur';
}
?>
```

Nous pouvons également définir une relation plus spécifique en utilisant un tableau :

```
<?php

class Profil extends AppModel {
    var $name = 'Profil';
    var $belongsTo = array(
        'Utilisateur' => array(
            'className'    => 'Utilisateur',
            'foreignKey'    => 'utilisateur_id'
        )
    );
}
?>
```

Les clés possibles pour le tableau d'association belongsTo sont :

- **className** : le nom de la classe du modèle que l'on souhaite associer au modèle actuel. Si l'on souhaite définir la relation 'Profil appartient à Utilisateur', la valeur associée à la clef 'className' devra être 'Utilisateur'.
- **foreignKey** : le nom de la clef étrangère que l'on trouve dans le modèle actuel. Ceci sera particulièrement pratique si vous avez besoin de définir des relations belongsTo multiples. La valeur par défaut de cette clef est le nom de l'autre modèle (avec des underscores) suffixé avec '_id'.
- **conditions** : un fragment de code SQL utilisé pour filtrer les enregistrements du modèle relié. C'est une bonne pratique que d'utiliser les noms des modèles dans ces portions de code : "Utilisateur.actif = 1" sera toujours mieux qu'un simple "actif = 1".
- **fields** : une liste des champs à récupérer lorsque les données du modèle associé sont parcourues. Par défaut, cela retourne tous les champs.
- **order** : un fragment SQL qui définit l'ordre des lignes de résultat retournées.
- **counterCache** : si il vaut true le Modèle associé incrémentera ou décrémentera automatiquement le champ "[nom_du_modele_au_singulier]_count" dans la table étrangère dès qu'un appel a save() ou delete() sera effectué. Si c'est une chaîne de caractère alors cela représente le nom du champ à utiliser. La valeur dans le champ du compteur représente le nombre d'enregistrements liés.
- **counterScope** : tableau de conditions optionnelles à utiliser pour mettre à jour le champ du cache de compteur.

Une fois que cette association aura été définie, les opérations de recherche sur le modèle Profil récupèreront également les enregistrements Utilisateurs liés s'il en existe :

```
//Exemple de résultats d'un appel à $this->Profil->find().

Array
(
   [Profil] => Array
        (
            [id] => 12
            [utilisateur_id] => 121
            [competences] => Cuisiner des gâteaux
            [created] => 2007-05-01 10:31:01
```

```
        )
    [Utilisateur] => Array
        (
            [id] => 121
            [nom] => Gwoo le Kungwoo
            [created] => 2007-05-01 10:31:01
        )
)
```

3.7.6.4 hasMany

Prochaine étape : définir une association "Utilisateur hasMany Commentaire". Une association hasMany nous permettra de récupérer les commentaires d'un utilisateur lors de la récupération d'un enregistrement Utilisateur.

Lorsque vous définissez les clés de votre base de données pour une relation de type hasMany, suivez cette convention :

hasMany: l'*autre* modèle contient la clé étrangère.

Relation	Schéma
Utilisateur hasMany Commentaire	Commentaire.utilisateur_id
Cake hasMany Vertue	Vertue.cake_id
Produit hasMany Option	Option.produit_id

On peut définir l'association hasMany dans notre modèle Utilisateur (/app/models/utilisateur.php) en utilisant une chaîne de caractère de cette manière :

```
<?php

class Utilisateur extends AppModel {
    var $name = 'Utilisateur ';
    var $hasMany = 'Commentaire';
}
?>
```

Nous pouvons également définir une relation plus spécifique en utilisant un tableau :

```
<?php

class Utilisateur extends AppModel {
    var $name = 'Utilisateur ';
    var $hasMany = array(
        'Commentaire' => array(
            'className'     => 'Commentaire',
            'foreignKey'    => 'utilisateur_id',
            'conditions'    => array('Commentaire.statut' => '1'),
            'order'    => 'Commentaire.created DESC',
            'limit'        => '5',
            'dependent'=> true
        )
    );
}
```

```
?>
```

Les clés possibles pour les tableaux d'association hasMany sont :

- **className**: le nom de la classe du modèle que l'on souhaite associer au modèle actuel. Si l'on souhaite définir la relation 'Utilisateur a plusieurs Commentaire', la valeur associée à la clef 'className' devra être 'Commentaire'.
- **foreignKey**: le nom de la clé etrangère que l'on trouve dans l'autre modèle. Ceci sera particulièrement pratique si vous avez besoin de définir des relations hasMany multiples. La valeur par défaut de cette clef est le nom du modèle actuel (avec des underscores) suffixé avec '_id'.
- **conditions**: un fragment de code SQL utilisé pour filtrer les enregistrements du modèle relié. C'est une bonne pratique que d'utiliser les noms des modèles dans ces portions de code : "Commentaire.statut= 1" sera toujours mieux qu'un simple "statut = 1".
- **fields**: une liste des champs à récupérer lorsque les données du modèle associé sont parcourues. Par défaut, cela retourne tous les champs.
- **order**: un fragment de code SQL qui définit l'ordre des entrées associées.
- **limit**: le nombre maximum d'entrées associées qui seront retournées.
- **offset**: le nombre d'entrées associées à sauter (les conditions et l'ordre de classement étant donnés) avant de récupérer de nouveaux enregistrements et de les associer.
- **dependent**: lorsque dependent vaut true, une suppression récursive du modèle est possible. Dans cet exemple, les enregistrements Commentaires seront supprimés lorsque leur Utilisateur associé l'aura été.
- **exclusive**: Lorsque exclusive est fixé à *true*, la suppression récursive de modèle 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 modèle. 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 modèle associé, utilisez le marqueur spécial `{$__cakeID__$}` dans la requête. Par exemple, si votre modèle 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 modèle Utilisateur récupèreront également les Commentaires reliés si ils existent :

```
//Exemple de résultats d'un appel à $this->Utilisateur->find().

Array
(
    [Utilisateur] => Array
        (
            [id] => 121
            [nom] => Gwoo le Kungwoo
            [created] => 2007-05-01 10:31:01
        )
    [Commentaire] => Array
        (
            [0] => Array
                (
                    [id] => 123
```

```
                    [utilisateur_id] => 121
                    [titre] => Sur Gwoo le Kungwoo
                    [corps] => La Kungwooité est assez Gwooteuse
                    [created] => 2006-05-01 10:31:01
                )
            [1] => Array
                (
                    [id] => 123
                    [utilisateur_id] => 121
                    [titre] => Plus sur Gwoo
                    [corps] => Mais qu'en est-il ?
                    [created] => 2006-05-01 10:41:01
                )
        )
)
```

Une chose dont il faut se rappeler est que vous aurez besoin d'une association "Commentaire belongsTo Utilisateur" 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 Commentaire depuis l'Utilisateur. En ajoutant l'association "Commentaire belongsTo Utilisateur" dans le modèle Commentaire, vous aurez la possibilité de connaître les données de l'Utilisateur depuis le modèle Commentaire - cela complète la connexion entre eux et permet un flot d'informations depuis n'importe lequel des deux modèles.

3.7.6.5 hasAndBelongsToMany (HABTM)

Très bien. A ce niveau, vous pouvez déjà vous considérer comme un professionnel des associations de modèles 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 modèles 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 modèles n'est pas exclusif dans le cadre d'une relation HABTM. Par exemple, relions notre modèle Recette avec un modèle Tag en utilisant HABTM. Le fait d'attacher le tag "Italien" à la recette de Gnocchi de ma grand-mère ne "consomme" pas le tag. Je peux aussi taguer mes Spaghettis Caramélisées au miel comme "Italien" si je le souhaite.

Les liens entre des objets liés par une association hasMany sont exclusif. Si mon Utilisateur "hasMany" Commentaires, un commentaire ne sera lié qu'à un utilisateur 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 modèles 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 modèles 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 modèle.

HABTM Nécessite une table de jointure séparée qui contient les deux noms de *modèles*.

Relation	Schéma (table HABTM en gras)
Recette HABTM Tag	**recettes_tags**.id, **recettes_tags**.recette_id, **recettes_tags**.tag_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

Le nom des tables est par convention dans l'ordre alphabétique.

Une fois que cette nouvelle table a été créée, on peut définir l'association HABTM dans les fichiers de modèle. Cette fois ci, nous allons directement voir la syntaxe tabulaire :

```
<?php

class Recette extends AppModel {
    var $name = 'Recette';
    var $hasAndBelongsToMany = array(
        'Tag' =>
            array(
                 'className'              => 'Tag',
                 'joinTable'              => 'recettes_tags',
                 'with'                   => '',
                'foreignKey'             => 'recette_id',
                'associationForeignKey'  => 'tag_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 modèle que l'on souhaite associer au modèle actuel. Si l'on souhaite définir la relation 'Utilisateur HABTM Commentaire', la valeur associée à la clef 'className' devra être 'Commentaire'.
- **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 modèle pour la table de jointure. Par défaut CakePHP créera automatiquement un modèle pour vous. Dans l'exemple ci-dessus la valeur aurait été RecettesTag. En utilisant cette clé vous pouvez surcharger ce nom par défaut. Le modèle de la table de jointure peut être utilisé comme tout autre modèle "classique" pour accéder directement à la table de jointure.
- **foreignKey**: le nom de la clef étrangère que l'on trouve dans le modèle 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 modèle actuel (avec des underscores) suffixé avec '_id'.
- **associationForeignKey**: le nom de la clé etrangère que l'on trouve dans l'autre modèle. Ceci sera particulièrement pratique si vous avez besoin de définir des relations HABTM multiples. La valeur

par défaut de cette clef est le nom de l'autre modèle (avec des underscores) suffixé avec '_id'.
- **unique**: Si *true* (valeur par défaut) Cake supprimera d'abord les enregistrement des relations existantesdans 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.
- **conditions**: un fragment de code SQL utilisé pour filtrer les enregistrements du modèle relié. C'est une bonne pratique que d'utiliser les noms des modèles dans ces portions de code : "Commentaire.statut= 1" sera toujours mieux qu'un simple "statut = 1".
- **fields**: une liste des champs à récupérer lorsque les données du modèle associé sont parcourues. Par défaut, cela retourne tous les champs.
- **order**: un fragment de code SQL qui définit l'ordre des entrées associées.
- **limit**: le nombre maximum d'entrées associées qui seront retournées.
- **offset**: le nombre d'entrées associées à sauter (les conditions et l'ordre de classement étant donnés) avant de récupérer de nouveaux enregistrements et de les associer.
- **finderQuery**: une requête SQL complète que CakePHP peut utiliser pour retrouver, supprimer ou créer de nouveaux enregistrements d'associations de modèles. Ceci ne devrait être utilisé que dans les situations qui nécessitent des résultats très personnalisés.

Une fois que cette association a été définie, les opérations de recherche sur le modèle Recette récupèreront également les Tag reliés si ils existent :

```
//Exemple de résultats d'un appel a $this->Recette->find().

Array
(
    [Recette] => Array
        (
            [id] => 2745
            [nom] => Bombes de sucres au chocolat glacé
            [created] => 2007-05-01 10:31:01
            [utilisateur_id] => 121
        )
    [Tag] => Array
        (
            [0] => Array
                (
                    [id] => 123
                    [nom] => Petit déjeuner
                )
           [1] => Array
                (
                    [id] => 124
                    [nom] => Dessert
                )
           [2] => Array
                (
                    [id] => 125
                    [nom] => Déprime sentimentale
                )
        )
)
```

N'oubliez pas de définir une association HABTM dans le modèle Tag si vous souhaitez retrouver les données de Recette lorsque vous manipulez le modèle Tag.

Il est également possible d'exécuter des requêtes de recherche personnalisées basées sur des relations HABTM. Regardez les exemples suivants :

En supposant que nous avons la même structure que dans les exemples ci-dessus (Recette HABTM Tag), disons que nous voulions récupérer toutes les Recettes avec le Tag "Dessert". Une solution rapide (mais incorrecte) pour faire ceci serait d'utiliser une condition complexe sur le modèle Recette :

```
$this->Recette->bindModel(array(
    'hasAndBelongsToMany' => array(
        'Tag' => array('conditions'=>array('Tag.nom'=>'Dessert'))
)));
$this->Recette->find('all');
```

```
// Données retournées
Array
(
    0 => Array
        {
        [Recette] => Array
            (
                [id] => 2745
                [nom] => Bombes de sucres au chocolat glacé
                [created] => 2007-05-01 10:31:01
                [utilisateur_id] => 121
            )
        [Tag] => Array
            (
               [0] => Array
                    (
                        [id] => 124
                        [nom] => Dessert
                    )
            )
    )
    1 => Array
        {
        [Recette] => Array
            (
                [id] => 2745
                [nom] => Gâteau au crabe
                [created] => 2008-05-01 10:31:01
                [utilisateur_id] => 121
            )
        [Tag] => Array
            (
            }
        }
}
```


Notez que cet exemple retourne TOUTES les recettes, mais seulement les tags "Dessert". Pour parvenir proprement à notre but, il y a de nombreux moyens de faire. Une option est de faire une recherche sur le modèle Tag (au lieu de Recette), ce qui nous donnera également toutes les Recettes associées.

```
$this->Recette->Tag->find('all',
array('conditions'=>array('Tag.nom'=>'Dessert')));
```

Nous pouvons également utiliser le modèle de la table de jointure (que cakePHP nous fournit), pour rechercher un ID donné.

```
$this->Recette->bindModel(array('hasOne' => array('RecettesTag')));
$this->Recette->find('all', array(
        'fields' => array('Recette.*'),
        'conditions'=>array('RecettesTag.tag_id'=>124) // id de Dessert
));
```

Il est également possible de créer une association exotique dans le but de créer autant de jointures que nécessaires pour permettre le filtrage, par exemple :

```
$this->Recette->bindModel(array(
    'hasOne' => array(
        'RecettesTag',
        'FiltreTag' => array(
            'className' => 'Tag',
            'foreignKey' => false,
            'conditions' => array('FiltreTag.id = RecettesTag.id')
))));
$this->Recette->find('all', array(
        'fields' => array('Recette.*'),
        'conditions'=>array('FiltreTag.nom'=>'Dessert')
));
```

Ces deux méthodes retourneront les données suivantes :

```
// Données retournées
Array
(
    0 => Array
        {
        [Recette] => Array
            (
                [id] => 2745
                [nom] => Bombes de sucres au chocolat glacé
                [created] => 2007-05-01 10:31:01
                [utilisateur_id] => 121
            )
    [Tag] => Array
        (
            [0] => Array
                (
                    [id] => 123
                    [nom] => Petit déjeuner
                )
           [1] => Array
                (
                    [id] => 124
                    [nom] => Dessert
                )
           [2] => Array
                (
                    [id] => 125
                    [nom] => Déprime sentimentale
                )
        )
}
```

La même astuce de lien peut être utilisée pour paginer facilement vos modèles HABTM. Juste un mot d'avertissement : comme paginate nécessite deux requêtes (une pour compter les enregistrements et une pour

récupérer les données effectives), soyez sÃ»rs d'avoir fourni le paramètre `false` à `bindModel();` car cela permet essentiellement de dire à CakePHP de garder l'association de manière persistance sur les requêtes multiples, au lieu de sur une seule comme dans le comportement par défaut. Merci de vous référer à l'API pour plus de détails.

Pour plus d'informations sur l'association de modèles à la volée lisez Créer et détruire des associations à la volée

Mélangez et faites correspondre les techniques pour parvenir à votre but !

3.7.6.6 hasMany through (The Join Model)

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

It is sometimes desirable to store additional data with a many to many association. Consider the following

Student hasAndBelongsToMany Course Course hasAndBelongsToMany Student

In other words, a Student can take many Courses and a Course can be taken my many Students. This is a simple many to many association demanding a table such as this

```
id | student_id | course_id
```

Now what if we want to store the number of days that were attended by the student on the course and their final grade? The table we'd want would be

```
id | student_id | course_id | days_attended | grade
```

The trouble is, hasAndBelongsToMany will not support this type of scenario because when hasAndBelongsToMany associations are saved, the association is deleted first. You would lose the extra data in the columns as it is not replaced in the new insert.

The way to implement our requirement is to use a **join model**, otherwise known (in Rails) as a **hasMany through** association. That is, the association is a model itself. So, we can create a new model CourseMembership. Take a look at the following models.

```
        student.php

        class Student extends AppModel
        {
            public $hasMany = array(
                'CourseMembership'
            );

            public $validate = array(
                'first_name' => array(
                    'rule' => 'notEmpty',
                    'message' => 'A first name is required'
                ),
                'last_name' => array(
                    'rule' => 'notEmpty',
```

```
                'message' => 'A last name is required'
            )
        );
    }

    course.php

    class Course extends AppModel
    {
        public $hasMany = array(
            'CourseMembership'
        );

        public $validate = array(
            'name' => array(
                'rule' => 'notEmpty',
                'message' => 'A course name is required'
            )
        );
    }

    course_membership.php

    class CourseMembership extends AppModel
    {
        public $belongsTo = array(
            'Student', 'Course'
        );

        public $validate = array(
            'days_attended' => array(
                'rule' => 'numeric',
                'message' => 'Enter the number of days the student attended'
            ),
            'grade' => array(
                'rule' => 'notEmpty',
                'message' => 'Select the grade the student received'
            )
        );
    }
```

The CourseMembership join model uniquely identifies a given Student's participation on a Course in addition to extra meta-information.

Working with join model data

Now that the models have been defined, let's see how we can save all of this. Let's say the Head of Cake School has asked us the developer to write an application that allows him to log a student's attendance on a course with days attended and grade. Take a look at the following code.

```
    controllers/course_membership_controller.php

    class CourseMembershipsController extends AppController
    {
        public $uses = array('CourseMembership');

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

        public function add() {
```

```
        if (! empty($this->data)) {

            if ($this->CourseMembership->saveAll(
                $this->data, array('validate' => 'first'))) {


                $this->redirect(array('action' => 'index'));
            }
        }
    }
}
```

views/course_memberships/add.ctp

```
<?php echo $form->create('CourseMembership'); ?>
    <?php echo $form->input('Student.first_name'); ?>
    <?php echo $form->input('Student.last_name'); ?>
    <?php echo $form->input('Course.name'); ?>
    <?php echo $form->input('CourseMembership.days_attended'); ?>
    <?php echo $form->input('CourseMembership.grade'); ?>
    <button type="submit">Save</button>
<?php echo $form->end(); ?>
```

You can see that the form uses the form helper's dot notation to build up the data array for the controller's save which looks a bit like this when submitted.

```
    Array
    (
        [Student] => Array
            (
                [first_name] => Joe
                [last_name] => Bloggs
            )

        [Course] => Array
            (
                [name] => Cake
            )

        [CourseMembership] => Array
            (
                [days_attended] => 5
                [grade] => A
            )

    )
```

Cake will happily be able to save the lot together and assigning the foreign keys of the Student and Course into CourseMembership with a saveAll call with this data structure. If we run the index action of our CourseMembershipsController the data structure received now from a find('all') is:

```
    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
                        )

                )

        )
```

There are of course many ways to work with a join model. The version above assumes you want to save everything at-once. There will be cases where you want to create the Student and Course independently and at a later point associate the two together with a CourseMembership. So you might have a form that allows selection of existing students and courses from picklists or ID entry and then the two meta-fields for the CourseMembership, e.g.

```
    views/course_memberships/add.ctp

    <?php echo $form->create('CourseMembership'); ?>
        <?php echo $form->input('Student.id', array('type' => 'text', 'label' =>
'Student ID', 'default' => 1)); ?>
        <?php echo $form->input('Course.id', array('type' => 'text', 'label' =>
'Course ID', 'default' => 1)); ?>
        <?php echo $form->input('CourseMembership.days_attended'); ?>
        <?php echo $form->input('CourseMembership.grade'); ?>
        <button type="submit">Save</button>
    <?php echo $form->end(); ?>
```

And the resultant POST

```
        Array
        (
            [Student] => Array
                (
                    [id] => 1
                )

            [Course] => Array
                (
                    [id] => 1
                )

            [CourseMembership] => Array
                (
```

```
                    [days_attended] => 10
                    [grade] => 5
                )

        )
```

Again Cake is good to us and pulls the Student id and Course id into the CourseMembership with the saveAll.

Join models are pretty useful things to be able to use and Cake makes it easy to do so with its built-in hasMany and belongsTo associations and saveAll feature.

3.7.6.7 Créer et détruire des Associations à la volée

Quelquefois il devient nécessaire de créer et détruire les associations de modèles 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 se font en utilisant les méthodes de modèles CakePHP bindModel() et unbindModel(). Mettons en place quelques modèles pour pouvoir ensuite voir comment fonctionnent bindModel() et unbindModel(). Nous commencerons avec deux modèles :

```php
<?php

class Meneur extends AppModel {
    var $name = 'Meneur';

    var $hasMany = array(
        'Suiveur' => array(
            'className' => 'Suiveur',
            'order'     => 'Suiveur.rang'
        )
    );
}

?>

<?php

class Suiveur extends AppModel {
    var $name = 'Suiveur';
}

?>
```

Maintenant, dans le contrôleur MeneursController, nous pouvons utiliser la méthode find() du modèle Meneur pour retrouver un Meneur et les Suiveurs associés. Comme vous pouvez le voir ci-dessus, le tableau d'association dans le modèle Meneur définit une relation "Meneur hasMany (a plusieurs) Suiveurs". Dans un but démonstratif, utilisons unbindModel() pour supprimer cette association dans une action du contrôleur.

```php
function uneAction() {
    // Ceci récupère tous les Meneurs, ainsi que leurs Suiveurs
    $this->Meneur->findAll();
```

```
    // Supprimons la relation hasMany() ...
    $this->Meneur->unbindModel(
        array('hasMany' => array('Suiveur'))
    );

    // Désormais l'utilisation de la fonction find() retournera
    // des Meneurs, sans aucun Suiveurs
    $this->Meneur->findAll();

    // 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() après unbindModel(),
    // ainsi cette ligne récupèrera une fois encore les Meneurs
    // avec leurs Suiveurs ...
    $this->Meneur->findAll();
}
```

Encore un rappel. Enlever ou ajouter des associations en utilisant bindModel() et unbindModel() ne fonctionne que pour la *prochaine* opération sur le modèle, à moins que le second paramètre n'ait été fixé à true. Si le second paramètre a été fixé à *true*, le lien reste en place pour la suite de la requête.

Voici un exemple basique d'utilisation de unbindModel() :

```
$this->Modele->unbindModel(
    array('associationType' => array('nomDeClasseModeleAssocie'))
);
```

Maintenant que nous sommes arrivés à supprimer une association à la volée, ajoutons-en une. Notre Meneur jusqu'à présent sans Principes a besoin d'être associé à quelques Principes. Le fichier de modèle pour notre modèle Principe est dépouillé, il n'y a que la ligne var $name. Associons à la volée des Principes à notre Meneur (mais rappelons-le, seulement pour la prochaine opération find). Cette fonction apparaît dans le contrôleur MeneursController :

```
function uneAutreAction() {
    // Il n'y a pas d'association Meneur hasMany Principe
    // dans le fichier de modèle meneur.php, ainsi un find
    // situé ici ne récupèrera que les Meneurs.
    $this->Meneur->findAll();

    // Utilisons bindModel() pour ajouter une nouvelle association
    // au modèle Meneur :
    $this->Meneur->bindModel(
        array('hasMany' => array(
                'Principe' => array(
                    'className' => 'Principe'
                )
            )
        )
    );

    // Maintenant que nous les avons associés correctement,
    // nous pouvons utiliser la fonction find une seule fois
    // pour récupérer les Meneurs avec leurs Principes associés :
```

```
    $this->Meneur->findAll();
}
```

Ã a 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->Modele->bindModel(
        array('nomAssociation' => array(
                'nomDeClasseModeleAssocie' => array(
                    // les clés normales d'une association sont à mettre ici
...
                )
            )
        )
    );
```

Bien que le modèle nouvellement associé n'ait besoin d'aucune définition d'association dans son fichier de modèle, il devra tout de même contenir les clés afin que la nouvelle association fonctionne bien.

3.7.6.8 Relations multiples avec le même modèle

Il y a des cas où un Modèle a plus d'une relation avec un autre Modèle. Par exemple, vous pourriez avoir un modèle Message qui a deux relations avec le modèle Utilisateur. Une relation avec l'utilisateur qui envoie un message et une seconde avec l'utilisateur qui reçoit le message. La table messages aura un champ utilisateur_id, mais aussi un champ receveur_id. Maintenant, votre modèle Message peut ressembler à quelque chose comme :

```
<?php
class Message extends AppModel {
    var $name = 'Message';
    var $belongsTo = array(
        'Expediteur' => array(
            'className' => 'Utilisateur',
            'foreignKey' => 'utilisateur_id'
        ),
        'Receveur' => array(
            'className' => 'Utilisateur',
            'foreignKey' => 'receveur_id'
        )
    );
}
?>
```

Receveur est un alias pour le modèle Utilisateur. Maintenant, voyons à quoi devrait ressembler le modèle Utilisateur.

```
<?php
class Utilisateur extends AppModel {
    var $name = 'Utilisateur';
    var $hasMany = array(
        'MessageEnvoye' => array(
            'className' => 'Message',
            'foreignKey' => 'utilisateur_id'
        ),
        'MessageRecu' => array(
```

```
            'className' => 'Message',
            'foreignKey' => 'receveur_id'
        )
    );
}
?>
```

3.7.6.9 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 modèles 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.

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);
```

Notez que les tableaux 'join' ne sont pas indexés.

Dans l'exemple ci-dessus, un modèle appelé Item est joint à gauche à la table channels. Vous pouvez ajouter un alias à la table, avec le nom du Modèle, ainsi les données retournées se conformeront à la structure de données de CakePHP.

Les clés qui définissent la jointure sont les suivants :

- **table** : la table pour la jointure.
- **alias** : un alias vers la table. Le nom du modèle 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 modèle relié :

```
$options['joins'] = array(
    array('table' => 'channels',
        'alias' => 'Channel',
        'type' => 'LEFT',
        $conditions = array(
            'Channel.id = Item.channel_id',
        )
    )
);

$options['conditions'] => array(
    'Channel.prive' => 1
)

$itemsPrives = $Item->find('all', $options);
```

Au besoin, vous pourriez réaliser plusieurs jointures dans une hasAndBelongsToMany :

Supposez une association Livre hasAndBelongsToMany Tag. Cette relation utilise une table livres_tags comme table de jointure, donc vous avez besoin de joindre la table livres à la table livres_tags et celle-ci avec la table tags :

```
$options['joins'] = array(
    array('table' => 'livres_tags',
        'alias' => 'LivresTag',
        'type' => 'inner',
        'conditions' => array(
            'Livres.id = LivresTag.livres_id'
        )
    ),
    array('table' => 'tags',
        'alias' => 'Tag',
        'type' => 'inner',
        'conditions' => array(
            'LivresTag.tag_id = Tag.id'
        )
    )
);

$options['conditions'] = array(
    'Tag.tag' => 'Nouvelle'
);

$livres = $Livre->find('all', $options);
```

Utiliser joins avec le comportement Containable pourrait conduire à quelques erreurs SQL (tables dupliquées), vous devez donc utiliser la méthode joins comme une alternative à Containable, si l'objectif principal est de réaliser des recherches basées sur les données liées. Containable est plus approprié pour restreindre le volume de données reliées rapportées par une instruction find .

3.7.7 Méthodes de Callbacks du Modèle

Si vous voulez glisser un bout de logique applicative juste avant ou après une opération d'un modèle CakePHP, utilisez les callbacks de modèle. Ces fonctions peuvent être définies dans les classes de modèle (cela comprend également votre classe AppModel). Notez bien les valeurs de retour attendues pour chacune de ces méthodes spéciales.

3.7.7.1 beforeFind

`beforeFind(mixed $donneesRequete)`

Appelée avant toute opération liée à la recherche. Les `$donneesRequete` 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 `$donneesRequete`), retournez *false*. Autrement, retournez la variable `$donneesRequete` 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.

3.7.7.2 afterFind

`afterFind(array $resultats, bool $primaire`

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 modèle, càd quelque chose come :

```
resultats = array(
  0 => array(
    'NomModele' => 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.

Si $primaire est faux, le format de $resultats sera un peu différent de ce que l'on peut attendre; à la place du résultat que vous auriez habituellement reçu d'une opération de recherche, vous aurez ceci :

```
resultats = array(
  'champ_1' => 'valeur',

  'champ_2' => 'valeur2'
);
```

Un code nécessitant que `$primaire` soit vrai auront 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 utilisé pour formater des dates.

```
function afterFind($resultats) {
    foreach ($resultats as $clef => $val) {
        if (isset($val['Evenement']['debut'])) {
            $results[$clef]['Evenement']['fin'] =
$this->dateFormatAfterFind($val['Evenement']['debut']);
```

```
        }
    }
    return $resultats;
}

function dateFormatAfterFind($date) {
    return date('d-m-Y', strtotime($date));
}
```

3.7.7.3 beforeValidate

beforeValidate()

Utilisez ce rappel pour modifier les données du modèle 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.

3.7.7.4 beforeSave

beforeSave()

Placez toute logique de pré-enregistrement dans cette fonction. Cette fonction s'exécute immediatement après que les données du modèle 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 sÃ»r, ceci peut être très facilement modifié. Utilisez le code ci-dessous dans le modèle approprié.

```
function beforeSave() {
    if(!empty($this->data['Evenement']['date_debut']) &&
!empty($this->data['Evenement']['date_fin'])) {
            $this->data['Evenement']['date_debut'] =
$this->dateFormatBeforeSave($this->data['Evenement']['date_debut']);
            $this->data['Evenement']['date_fin'] =
$this->dateFormatBeforeSave($this->data['Evenement']['date_fin']);
    }
    return true;
}

function dateFormatBeforeSave($dateString) {
    return date('Y-m-d', strtotime($dateString)); // Le sens d'affichage provient de
là
}
```

Assurez-vous que beforeSave() retourne vrai ou bien votre sauvegarde échouera.

3.7.7.5 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).

3.7.7.6 beforeDelete

`beforeDelete(boolean $cascade)`

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.

3.7.7.7 afterDelete

`afterDelete()`

Placez dans cette méthode de rappel, toute logique que vous souhaitez exécuter après chaque suppression.

3.7.7.8 onError

`onError()`

Appelée si quelque problème se produit.

3.7.8 Attributs des Modèles

Les attributs du Modèle vous permettent de définir des propriétés qui peuvent modifier son comportement par défaut.

Pour une liste complète des attributs du modèle et leurs descriptions, visitez l'API CakePHP : http://api.cakephp.org/class/model.

3.7.8.1 useDbConfig

La propriété `useDbConfig` spécifie quel paramètre du fichier de configuration de la base de données vous voulez utiliser. Le fichier de configuration de la base de données est stocké dans **`/app/config/database.php`**.

Exemple d'utilisation:

```
class Example extends AppModel {
    var $useDbConfig = 'alternative';
}
```

La valeur par défaut de `useDbConfig` est 'default'.

3.7.8.2 useTable

La propriété `useTable` spécifie le nom de la table de la base de données. Par défaut, le modèle utilise le nom de la classe modèle au pluriel et en minuscule. Donnez à cet attribut le nom d'une table alternative, ou `false`

Exemple d'utilisation :

```
class Exemple extends AppModel {
   var $useTable = false; // Ce modèle n'utilise pas de table de la base de
données
}
```

Table alternative :

```
class Exemple extends AppModel {
   var $useTable = 'exmp'; // Ce modèle utilise la table 'exmp'
}
```

3.7.8.3 tablePrefix

Le nom du préfixe de la table utilisée par le modèle. Initialement, le préfixe utilisé est celui renseigné avec la connexion à la base de données dans **`/app/config/database.php`**. Par défaut il n'y a aucun préfixe. Vous pouvez surcharger la valeur par défaut en configurant l'attribut `tablePrefix` dans le modèle.

Exemple d'utilisation :

```
class Exemple extends AppModel {
   var $tablePrefix = 'alt_'; // cherchera la table 'alt_exemples'
}
```

3.7.8.4 primaryKey

Chaque table a normalement une clé primaire, `id`. Vous pouvez changer le champ qui sera utilisé par le modèle comme sa clé primaire. Ceci est fréquent lorsque l'on configure CakePHP pour utiliser une table existant déjà dans la base de données.

Exemple d'utilisation :

```
class Exemple extends AppModel {
    var $primaryKey = 'exemple_id'; // exemple_id est le nom du champ dans la base de
données
}
```

3.7.8.5 displayField

L'attribut `displayField` spécifie quel champ de la base de données doit être utilisé comme intitulé pour l'enregistrement. L'intitulé est utilisé dans le maquettage rapide (scaffolding) ainsi que dans les appels à `find('list')`. Le modèle utilisera `name` ou `title` par défaut.

Par exemple, pour utiliser le champ `pseudo` :

```
class Utilisateur extends AppModel {
   var $displayField = 'pseudo';
}
```

Les noms de champs multiples ne peuvent pas être combinés en un unique champ à afficher. Par exemple, vous ne pouvez pas spécifier `array('prenom', 'nom')` comme le champ à afficher.

3.7.8.6 recursive

La propriété recursive définit la profondeur jusqu'à laquelle CakePHP doit récupérer les données des modèles associés, via les méthodes `find()`, `findAll()` et `read()`.

Imaginons que votre application représente des Groupes qui appartiennent à un domaine et qui ont plusieurs Utilisateurs qui, à leur tour, ont plusieurs Articles. Vous pouvez attribuer à $recursive des valeurs différentes en fonction de la quantité de données que vous souhaitez récupérer en appelant $this->Groupe->find() :

Profondeur	Description
-1	Cake récupèrera seulement les données de Groupe, aucune jointure.
0	Cake récupèrera les données de Groupe ainsi que son domaine
1	Cake récupèrera un Groupe, son domaine et les Utilisateurs associés
2	Cake récupèrera un Groupe, son domaine, les Utilisateurs associés, et les Articles associés aux Utilisateurs

Ne le fixer pas trop haut par rapport à vos besoins. Demander à CakePHP de récupérer des données dont vous n'avez pas besoin ralentit votre application inutilement.

Si vous voulez combiner $recursive avec la fonctionnalité `fields`, vous devrez ajouter manuellement les colonnes contenant les clés étrangères nécessaires dans le tableau `fields`. Dans l'exemple ci-dessus, cela pourrait être l'ajout de `domaine_id`

3.7.8.7 order

L'ordre par défaut des données lors de toute opération find. Les valeurs possibles sont :

```
var $order = "champ"
var $order = "Modele.champ";
var $order = "Modele.champ asc";
var $order = "Modele.champ ASC";
var $order = "Modele.champ DESC";
var $order = array("Modele.champ" => "asc", "Modele.champ2" => "DESC");
```

3.7.8.8 data

Contient des méta-données décrivant les champs de la table de la base de données associée au modèle. Chaque champ est décrit par :

- nom
- type (integer, string, datetime, etc.)

- null
- valeur par défaut
- longueur

Exemple d'utilisation :

```
var $_schema = array(
    'prenom' => array(
        'type' => 'string',
        'length' => 30
    ),
    'nom' => array(
        'type' => 'string',
        'length' => 30
    ),
    'email' => array(
        'type' => 'string',
        'length' => 30
    ),
    'message' => array(
        'type' => 'text'
    )
);
```

3.7.8.10 validate

Cet attribut rassemble les règles permettant au modèle de décider de la validité des données avant une sauvegarde. Les clés mentionnées après le champ contiennent des expressions régulières permettant au modèle d'essayer de faire des correspondances.

Il n'est pas nécessaire d'appeler la méthode validate() avant save() : cette dernière validera automatiquement les données avant de sauvegarder de façon définitive.

Pour plus d'informations concernant la validation, regardez le chapitre Validation des données plus loin dans ce manuel

3.7.8.11 virtualFields

Tableau de champs virtuels que le modèle possède. Les champs virtuels sont des expressions SQL "aliasées". Les champs ajoutés à cette propriété seront lus comme les autres champs d'un modèle, mais ne seront pas enregistrables.

Exemple d'utilisation :

```
var $virtualFields = array(
    'nom' => 'CONCAT(Utilisateur.prenom, ' ', Utilisateur.nom)'
);
```

Dans les opérations de find ultérieures, vos résultats Utilisateur contiendrait une clé `nom` avec le résultat de la concaténation. Il n'est pas recommandé de créer des champs virtuels avec les mêmes noms que les colonnes de la base de données, ceci peut causer des erreurs SQL.

3.7.8.12 name

Comme vu plus tôt dans ce chapitre, l'attribut name est une caractéristique pour la compatibilité avec PHP4, il est fixé à la même valeur que le nom du modèle.

Exemple d'utilisation :

```
class Exemple extends AppModel {
   var $name = 'Exemple';
}
```

3.7.8.13 cacheQueries

Bien que les fonctions de modèle de CakePHP devraient vous emmener là où vous souhaitez aller, n'oubliez pas que les classes de modèles 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 modèle. Ce concept est souvent appelé *fat model* ("modèle gras").

```
class Exemple extends AppModel {

    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 contrôleur.

```
$recent = $this->Exemple->getRecent();
```

3.7.9.1 Using virtualFields

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Virtual fields are a new feature in the Model for CakePHP 1.3. Virtual fields allow you to create arbitrary SQL expressions and assign them as fields in a Model. These fields cannot be saved, but will be treated like other model fields for read operations. They will be indexed under the model's key alongside other model fields.

How to create virtual fields

Creating virtual fields is easy. In each model you can define a $virtualFields property that contains an array of `field => expressions`. An example of virtual field definitions would be:

```
var $virtualFields = array(
    'name' => 'CONCAT(User.first_name, " ", User.last_name)'
```

```
);
```

In subsequent find operations, your User results would contain a `name` key with the result of the concatenation. It is not advisable to create virtual fields with the same names as columns on the database, this can cause SQL errors.

Using virtual fields

Creating virtual fields is straightforward and easy, interacting with virtual fields can be done through a few different methods.

`Model::hasField()`

`Model::hasField()` has been updated so that it returns true if the model has a `virtualField` with the correct name. By setting the second parameter of `hasField` to true, `virtualFields` will also be checked when checking if a model has a field. Using the example field above,

```
$this->User->hasField('name'); // Will return false, as there is no concrete field
called name
$this->User->hasField('name', true); // Will return true as there is a virtual field
called name
```

`Model::isVirtualField()`

This method can be used to check if a field/column is a virtual field or a concrete field. Will return true if the column is virtual.

```
$this->User->isVirtualField('name'); //true
$this->User->isVirtualField('first_name'); //false
```

`Model::getVirtualField()`

This method can be used to access the SQL expression that comprises a virtual field. If no argument is supplied it will return all virtual fields in a Model.

```
$this->User->getVirtualField('name'); //returns 'CONCAT(User.first_name, ' ',
User.last_name)'
```

`Model::find()` and virtual fields

As stated earlier `Model::find()` will treat virtual fields much like any other field in a model. The value of a virtual field will be placed under the model's key in the resultset. Unlike the behavior of calculated fields in 1.2

```
$results = $this->User->find('first');

// results contains the following
array(
    'User' => array(
        'first_name' => 'Mark',
        'last_name' => 'Story',
```

```
        'name' => 'Mark Story',
        //more fields.
    )
);
```

Pagination and virtual fields

Since virtual fields behave much like regular fields when doing find's, `Controller::paginate()` has been updated to allows sorting by virtual fields.

3.7.10 Virtual fields

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Virtual fields are a new feature in the Model for CakePHP 1.3. Virtual fields allow you to create arbitrary SQL expressions and assign them as fields in a Model. These fields cannot be saved, but will be treated like other model fields for read operations. They will be indexed under the model's key alongside other model fields.

3.7.10.1 Creating virtual fields

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Creating virtual fields is easy. In each model you can define a `$virtualFields` property that contains an array of field => expressions. An example of a virtual field definition using MySQL would be:

```
var $virtualFields = array(
    'full_name' => 'CONCAT(User.first_name, " ", User.last_name)'
);
```

And with PostgreSQL:

```
var $virtualFields = array(
    'name' => 'User.first_name || \' \' || User.last_name'
);
```

In subsequent find operations, your User results would contain a `name` key with the result of the concatenation. It is not advisable to create virtual fields with the same names as columns on the database, this can cause SQL errors.

It is not always useful to have **User.first_name** fully qualified. If you do not follow the convention (i.e. you have multiple relations to other tables) this would result in an error. In this case it may be better to just use **first_name || \'\' || last_name** without the Model Name.

3.7.10.2 Using virtual fields

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Creating virtual fields is straightforward and easy, interacting with virtual fields can be done through a few different methods.

Model::hasField()

Model::hasField() has been updated so that it can return true if the model has a virtualField with the correct name. By setting the second parameter of hasField to true, virtualFields will also be checked when checking if a model has a field. Using the example field above,

```
$this->User->hasField('name'); // Will return false, as there is no concrete field
called name
$this->User->hasField('name', true); // Will return true as there is a virtual field
called name
```

Model::isVirtualField()

This method can be used to check if a field/column is a virtual field or a concrete field. Will return true if the column is virtual.

```
$this->User->isVirtualField('name'); //true
$this->User->isVirtualField('first_name'); //false
```

Model::getVirtualField()

This method can be used to access the SQL expression that comprises a virtual field. If no argument is supplied it will return all virtual fields in a Model.

```
$this->User->getVirtualField('name'); //returns 'CONCAT(User.first_name, ' ',
User.last_name)'
```

Model::find() and virtual fields

As stated earlier `Model::find()` will treat virtual fields much like any other field in a model. The value of a virtual field will be placed under the model's key in the resultset. Unlike the behavior of calculated fields in 1.2

```
$results = $this->User->find('first');

// results contains the following
array(
    'User' => array(
        'first_name' => 'Mark',
        'last_name' => 'Story',
        'name' => 'Mark Story',
        //more fields.
    )
);
```

Pagination and virtual fields

Since virtual fields behave much like regular fields when doing find's, `Controller::paginate()` has been updated to allow sorting by virtual fields.

3.7.10.3 Virtual fields and model aliases

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

When you are using virtualFields and models with aliases that are not the same as their name, you can run into problems as virtualFields do not update to reflect the bound alias. If you are using virtualFields in models that have more than one alias it is best to define the virtualFields in your model's constructor

```
function __construct($id = false, $table = null, $ds = null) {
    parent::__construct($id, $table, $ds);
    $this->virtualFields['name'] = sprintf('CONCAT(%s.first_name, " ", %s.last_name)',
$this->alias, $this->alias);
}
```

This will allow your virtualFields to work for any alias you give a model.

3.7.10.4 Limitations of virtualFields

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The implementation of `virtualFields` in 1.3 has a few limitations. First you cannot use `virtualFields` on associated models for conditions, order, or fields arrays. Doing so will generally result in an SQL error as the fields are not replaced by the ORM. This is because it's difficult to estimate the depth at which an associated model might be found.

A common workaround for this implementation issue is to copy `virtualFields` from one model to another at runtime when you need to access them.

```
$this->virtualFields['full_name'] = $this->Author->virtualFields['full_name'];
```

Alternatively, you can define `$virtualFields` in your model's constructor, using `$this->alias`, like so:

```
public function __construct($id=false,$table=null,$ds=null){
  parent::__construct($id,$table,$ds);
  $this->virtualFields = array(
    'name'=>"CONCAT(`{$this->alias}`.`first_name`,'
','`{$this->alias}`.`last_name`)"
  );
}
```

3.7.11 Transactions

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

To perform a transaction, a model's tables must be of a type that supports transactions.

All transaction methods must be performed on a model's DataSource object. To get a model's DataSource from within the model, use:

```
    $dataSource = $this->getDataSource();
```

You can then use the data source to start, commit, or roll back transactions.

```
    $dataSource->begin($this);

    //Perform some tasks

    if(/*all's well*/) {
        $dataSource->commit($this);
    } else {
        $dataSource->rollback($this);
    }
```

Nested transactions are currently not supported. If a nested transaction is started, a commit will return false on the parent transaction.

3.8 Comportements

Les comportements (*behaviors*) de Modèle sont une manière d'organiser certaines des fonctionnalités définies dans les modèles CakePHP. Ils nous permettent de séparer la logique qui ne doit pas être directement reliée à un modèle, mais qui nécessite d'être là. En offrant une simple, mais puissante, manière d'étendre les modèles, les comportements nous permettent d'attacher des fonctionnalités aux modèles en définissant une simple variable de classe. C'est comme çà que les comportements permettent de débarrasser les modèles 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 modèles et qui peut alors être extrapolé.

Par exemple, considérez un modèle qui nous donne accès à une table qui stocke des informations sur la structure d'un arbre hiérarchique. Supprimer, ajouter ou déplacer les noeuds 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 modèle de base (pour chaque modèle nécessitant cette fonctionnalité), nous pourrions simplement dire à notre modèle d'utiliser le Comportement Tree (*TreeBehavior*) ou, en des termes plus formels, nous dirions à notre modèle de se comporter comme un Arbre. On appelle cela attacher un comportement à un modèle. Avec une seule ligne de code, notre modèle CakePHP disposera d'un nouvel ensemble complet de méthodes lui permettant d'interagir avec la structure sous-jacente.

CakePHP contient déjà des comportements pour les structures en arbre, les contenus traduits, les interactions par liste de contrôle d'accès, sans oublier les comportements 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 comportements aux modèles, l'utilisation des comportements intégrés à CakePHP et la manière de créer nos propres comportements.

3.8.1 Utiliser les Comportements

Les Comportements sont attachés aux modèles grâce à la variable $actsAs des classes modèle :

```
<?php
class Categorie extends AppModel {
    var $name   = 'Categorie';
    var $actsAs = array('Tree');
```

```
}

?>
```

Cette exemple montre comme un modèle Catégorie pourrait être gérer dans une structure en arbre en utilisant le comportement Tree. Une fois qu'un comportement a été spécifié, utilisez les méthodes qu'il ajoute comme si elles avaient toujours existé et fait partie du modèle original :

```
// Définir ID
$this->Categorie->id = 42;

// Utiliser la méthode children() du comportement :
$enfants = $this->Categorie->children();
```

Quelques comportements peuvent nécessiter ou permettre des réglages quand ils sont attachés au modèle. Ici, nous indiquons à notre comportement Tree les noms des champs "left" et "right" de la table sous-jacente :

```
<?php

class Categorie extends AppModel {
    var $name   = 'Categorie';
    var $actsAs = array('Tree' => array(
        'left'  => 'noeud_gauche',
        'right' => 'noeud_droit'
    ));
}

?>
```

Nous pouvons aussi attacher plusieurs comportements à un modèle. Il n'y aucune raison pour que, par exemple, notre modèle Categorie se comporte seulement comme un arbre, il pourrait aussi supporter l'internationalisation :

```
<?php

class Categorie extends AppModel {
    var $name   = 'Categorie';
    var $actsAs = array(
        'Tree' => array(
          'left'  => 'noeud_gauche',
          'right' => 'noeud_droit'
        ),
        'Translate'
    );
}

?>
```

Jusqu'à présent, nous avons ajouter les comportements aux modèles en utilisant une variable de classe. Cela signifie que nos comportements seront attachés à nos modèles de tout au long de leur durée vie. Pourtant, nous pourrions avoir besoin de "détacher" les comportements des modèles à l'exécution. Considérons que dans notre précédent modèle Catégorie, lequel agit comme un modèle Tree et Translate, nous ayons besoin pour quelque raison de le forcer à ne plus agir comme un modèle Translate :

```
// Détache un comportement de notre modèle :
$this->Categorie->Behaviors->detach('Translate');
```

Cela fera que notre modèle Categorie arrêtera dorénavant de se comporter comme un modèle Translate. Nous pourrions avoir besoin, sinon, de désactiver simplement le comportement Translate pour qu'il n'agisse pas sur les opérations normales de notre modèle : nos *finds*, nos *saves*, etc. En fait, nous cherchons à désactiver le comportement qui agit sur nos callbacks de modèle CakePHP. Au lieu de détacher le comportement, nous allons dire à notre modèle d'arrêter d'informer ses callbacks du comportement Translate :

```
// Empêcher le comportement de manipuler nos callbacks de modèle
$this->Categorie->Behaviors->disable('Translate');
```

Nous pourrions également avoir besoin de chercher si notre comportement manipule ces callbacks de modèle et si ce n'est pas le cas, alors de restaurer sa capacité à réagir avec eux :

```
// Si notre comportement ne manipule pas nos callbacks de modèle
if (!$this->Categorie->Behaviors->enabled('Translate')) {
    // Disons lui de le faire maintenant !
    $this->Category->Behaviors->enable('Translate');
}
```

De la même manière que nous pouvons détacher complètement un comportement d'un modèle à l'exécution, nous pouvons aussi attacher de nouveaux comportements. Disons que notre modèle familier Categorie nécessite de se comporter comme un modèle de NoÃ«l, mais seulement le jour de NoÃ«l :

```
// Si nous sommes le 25 déc
if (date('m/d') == '12/25') {
    // Notre modèle nécessite de se comporter comme un modèle de NoÃ«l
    $this->Category->Behaviors->attach('Christmas');
}
```

Nous pouvons aussi utiliser la méthode attach pour réécrire les réglages du comportement :

```
// Nous changerons un réglage de notre comportement déjà
attaché
$this->Categorie->Behaviors->attach('Tree', array('left' =>
'nouveau_noeud_gauche'));
```

Il y a aussi une méthode pour obtenir la liste des comportements qui sont attachés à un modèle. Si nous passons le nom d'un comportement à une méthode, elle nous dira si ce comportement est attaché au modèle, sinon elle nous donnera la liste des comportements attachés :

```
// Si le comportement Translate n'est pas attaché
if (!$this->Categorie->Behaviors->attached('Translate')) {
    // Obtenir la liste de tous les comportements qui sont attachés au modèle
    $comportements = $this->Categorie->Behaviors->attached();
}
```

3.8.2 Créer des Comportements

Les Comportements qui sont attachés aux Modèles voient leurs *callbacks* appelés automatiquement. Ces *callbacks* sont similaires à ceux qu'on trouve dans les Modèles : beforeFind, afterFind, beforeSave, afterSave, beforeDelete, afterDelete et onError. Voir Méthodes de Callback.

C'est souvent pratique d'utiliser un comportement du coeur comme modèle pour créer les vôtres. Vous les trouverez dans `cake/libs/models/behaviors/`.

Chaque *callback* prend comme premier paramètre, une référence du modèle par lequel il est appelé.

En plus de l'implémentation des *callbacks*, vous pouvez ajouter des réglages par comportement et/ou par liaison d'un comportement au modèle. Des informations à propos des réglages spécifiques peuvent être trouvées dans les chapitres concernant les comportements du coeur et leur configuration.

Voici un exemple rapide qui illustre comment les réglages peuvent êtres passés du modèle au comportement :

```
class Post extends AppModel {
    var $name = 'Post'
    var $actsAs = array(
        'VotreComportement' => array(
        'cle_option1' => 'valeur_option1'
        )
    );
}
```

Depuis la version 1.2.8004, CakePHP ajoute ces réglages une seule fois par modèle/alias. Pour que vos comportements restent évolutifs, vous devriez respecter les alias (ou les modèles).

Une fonction de mise à jour facile, *setup*, devrait ressembler à quelque chose comme ça :

```
function setup(&$model, $settings) {
    if (!isset($this->settings[$model->alias])) {
    $this->settings[$model->alias] = array(
        'cle_option1' => 'valeur_defaut_option1',
        'cle_option2' => 'valeur_defaut_option2',
        'cle_option3' => 'valeur_defaut_option3',
    );
}
$this->settings[$Model->alias] = array_merge(
$this->settings[$Model->alias], (array)$settings);
}
```

3.8.3 Creating behavior methods

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Behavior methods are automatically available on any model acting as the behavior. For example if you had:

```
class Duck extends AppModel {
    var $name = 'Duck';
    var $actsAs = array('Flying');
}
```

You would be able to call FlyingBehavior methods as if they were methods on your Duck model. When creating behavior methods you automatically get passed a reference of the calling model as the first parameter. All other supplied parameters are shifted one place to the right. For example

```
$this->Duck->fly('toronto', 'montreal');
```

Although this method takes two parameters, the method signature should look like:

```
function fly(&$Model, $from, $to) {
    // Do some flying.
}
```

Keep in mind that methods called in a `$this->doIt()` fashion from inside a behavior method will not get the $model parameter automatically appended.

3.8.4 Behavior callbacks

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Model Behaviors can define a number of callbacks that are triggered before/after the model callbacks of the same name. Behavior callbacks allow your behaviors to capture events in attached models and augment the parameters or splice in additional behavior.

The available callbacks are:

3.8.5 Creating a behavior callback

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Model behavior callbacks are defined as simple methods in your behavior class. Much like regular behavior methods, they receive a `$Model` parameter as the first argument. This parameter is the model that the behavior method was invoked on.

`function beforeFind(&$model, $query)`

If a behavior's beforeFind returns false it will abort the find(). Returning an array will augment the query parameters used for the find operation.

`afterFind(&$model, $results, $primary)`

You can use the afterFind to augment the results of a find. The return value will be passed on as the results to either the next behavior in the chain or the model's afterFind.

`beforeDelete(&$model, $cascade = true)`

You can return false from a behavior's beforeDelete to abort the delete. Return true to allow it continue.

`afterDelete(&$model)`

You can use afterDelete to perform clean up operations related to your behavior.

beforeSave(&$model)

You can return false from a behavior's beforeSave to abort the save. Return true to allow it continue.

afterSave(&$model, $created)

You can use afterSave to perform clean up operations related to your behavior. $created will be true when a record is created, and false when a record is updated.

beforeValidate(&$model)

You can use beforeValidate to modify a model's validate array or handle any other pre-validation logic. Returning false from a beforeValidate callback will abort the validation and cause it to fail.

3.9 Sources de Données

Les Sources de données (*DataSources*) sont les liens entre les modèles 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 dbo_* dans **cake/libs/model/datasources/dbo/**), un résumé de ceux-ci est listé ici pour votre confort :

- dbo_adodb.php
- dbo_db2.php
- dbo_firebird.php
- dbo_mssql.php
- dbo_mysql.php
- dbo_mysqli.php
- dbo_odbc.php
- dbo_oracle.php
- dbo_postgres.php
- dbo_sqlite.php
- dbo_sybase.php

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 modèle. 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 **dbo_mysql.php** ou **dbo_mssql.php**) est plus sÃ»r.

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.

3.9.1 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` et `update`.

Méthodes qui doivent être implémentées :

- et au moins l'une de celles-ci :

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 modèle.

Et c'est à peu près tout ce qu'il y a dire ici. En couplant cette source de données à un modèle, vous êtes alors en mesure d'utiliser `Model::find()/save()`, 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 `'conditions'`, `'limit'` ou même vos paramètres personnalisés).

3.9.2 Un Exemple

Voici un exemple simple sur la manière d'utiliser les *Datasources* et `HttpSocket`, pour implémenter une source Twitter vraiment basique, qui permet de requêter l'API Twitter et de poster les nouvelles mises à jour du statut vers un compte configuré.

Cet exemple ne fonctionnera qu'avec PHP 5.2 et supérieur, à cause de l'usage de `json_decode` pour l'analyse des données au format JSON.

Vous pourriez placez la source de données Twitter dans **`app/models/datasources/twitter_source.php`** :

```
<?php
 * DataSource Twitter
 *
 * Utilisée pour lire et écrire sur Twitter, à travers les
modèles.
 *
 * PHP Version 5.x
 *
 * CakePHP(tm) : Rapid Development Framework (http://www.cakephp.org)
 * Copyright 2005-2009, Cake Software Foundation, Inc. (http://www.cakefoundation.org)
 *
 * Licensed under The MIT License
 * Redistributions of files must retain the above copyright notice.
 *
 * @filesource
 * @copyright     Copyright 2009, Cake Software Foundation, Inc.
(http://www.cakefoundation.org)
 * @link          http://cakephp.org CakePHP(tm) Project
 * @license       http://www.opensource.org/licenses/mit-license.php The MIT License
 */
App::import('Core', 'HttpSocket');
class TwitterSource extends DataSource {
    protected $_schema = array(
        'tweets' => array(
            'id' => array(
                'type' => 'integer',
                'null' => true,
                'key' => 'primary',
                'length' => 11,
            ),
```

```
            'text' => array(
                'type' => 'string',
                'null' => true,
                'key' => 'primary',
                'length' => 140
            ),
            'status' => array(
                'type' => 'string',
                'null' => true,
                'key' => 'primary',
                'length' => 140
            ),
        )
    );
    public function __construct($config) {
        $auth = "{$config['login']}:{$config['password']}";
        $this->connection = new HttpSocket(
            "http://{$auth}@twitter.com/"
        );
        parent::__construct($config);
    }
    public function listSources() {
        return array('tweets');
    }
    public function read($model, $queryData = array()) {
        if (!isset($queryData['conditions']['username'])) {
            $queryData['conditions']['username'] = $this->config['login'];
        }
        $url = "/statuses/user_timeline/";
        $url .= "{$queryData['conditions']['username']}.json";

        $response = json_decode($this->connection->get($url), true);
        $results = array();

        foreach ($response as $record) {
            $record = array('Tweet' => $record);
            $record['User'] = $record['Tweet']['user'];
            unset($record['Tweet']['user']);
            $results[] = $record;
        }
        return $results;
    }
    public function create($model, $fields = array(), $values = array()) {
        $data = array_combine($fields, $values);
        $result = $this->connection->post('/statuses/update.json', $data);
        $result = json_decode($result, true);
        if (isset($result['id']) && is_numeric($result['id'])) {
            $model->setInsertId($result['id']);
            return true;
        }
        return false;
    }
    public function describe($model) {
        return $this->_schema['tweets'];
    }
}
?>
```

Votre implémentation de modèle pourrait être aussi simple que :

```
<?php
class Tweet extends AppModel {
```

```
    public $useDbConfig = 'twitter';
}
?>
```

Si nous n'avions pas défini notre schéma dans la source de données elle-même, vous obtiendriez ici un message d'erreur.

Et les paramètres de configuration dans votre `app/config/database.php` devraient ressembler à quelque chose comme çà :

```
<?php
    var $twitter = array(
        'datasource' => 'twitter',
        'login' => 'username',
        'password' => 'password',
    );
?>
```

Utilisation des méthodes de modèle familières depuis un contrôleur :

```
<?php
// Utilisera le nom d'utilisateur défini dans $twitter, comme montré
ci-dessus :
$tweets = $this->Tweet->find('all');

// Trouve les tweets par un autre nom d'utilisateur
$conditions= array('username' => 'caketest');
$autresTweets = $this->Tweet->find('all', compact('conditions'));
?>
```

De la même façon, une nouvelle mise à jour du statut :

```
<?php
$this->Tweet->save(array('status' => 'Ceci est une mise à jour'));
?>
```

3.9.3 Plugin DataSources and Datasource Drivers

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Plugin Datasources

You can also package Datasources into plugins.

Simply place your datasource file into `plugins/[your_plugin]/models/datasources/[your_datasource]_source.php` and refer to it using the plugin notation:

```
var $twitter = array(
    'datasource' => 'Twitter.Twitter',
    'username' => 'test@example.com',
    'password' => 'hi_mom',
);
```

Plugin DBO Drivers

In addition, you can also add to the current selection of CakePHP's dbo drivers in plugin form.

Simply add your drivers to `plugins/[your_plugin]/models/datasources/dbo/[your_driver].php` and again use plugin notation:

```
var $twitter = array(
    'driver' => 'Twitter.Twitter',
    ...
);
```

Combining the Two

Finally, you're also able to bundle together your own DataSource and respective drivers so that they can share functionality. First create your main class you plan to extend:

```
plugins/[social_network]/models/datasources/[social_network]_source.php :
<?php
class SocialNetworkSource extends DataSource {
    // general functionality here
}
?>
```

And now create your drivers in a sub folder:

```
plugins/[social_network]/models/datasources/[social_network]/[twitter].php
<?php
class Twitter extends SocialNetworkSource {
    // Unique functionality here
}
?>
```

And finally setup your `database.php` settings accordingly:

```
var $twitter = array(
    'driver' => 'SocialNetwork.Twitter',
    'datasource' => 'SocialNetwork.SocialNetwork',
);
var $facebook = array(
    'driver' => 'SocialNetwork.Facebook',
    'datasource' => 'SocialNetwork.SocialNetwork',
);
```

Just like that, all your files are included **Automagically!** No need to place `App::import()` at the top of all your files.

3.10 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 Vues.

3.10.1 Templates de Vue

La couche vue 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.

Les fichiers de vues sont stockées dans /app/views/, dans un dossier portant le nom du contrôleur qui utilise ces fichiers et le nom de la vue correspondante. Par exemple, l'action voir() du contrôleur Produits devrait normalement se trouver dans /app/views/produits/voir.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 :

- **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.
- **elements** : morceaux de code de vue plus petits, réutilisables. Les éléments sont habituellement rendus dans les vues.
- **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.

3.10.2 Gabarits (layouts)

Un *layout* (gabarit) contient le code de présentation qui enveloppe une vue. Tout ce que vous voulez voir dans toutes vos vues devrait être placé dans un layout.

Les fichiers de gabarits devraient être placés dans /app/views/layouts. Le gabarit par défaut de CakePHP peut être surchargé, en créant un nouveau layout par défaut dans /app/views/layouts/default.ctp. Une fois qu'un nouveau layout par défaut a été créé, le code de rendu de vue du contrôleur est placé à l'intérieur de celui-ci lorsque la page est affichée.

Lorsque vous créez un gabarit, vous devez dire à CakePHP où placer le code de vos vues. Pour ce faire, assurez-vous que votre gabarit contient la variable $content_for_layout (et optionnellement, $title_for_layout). Voici un exemple de ce à quoi pourrait ressembler un layout par défaut :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title><?php echo $title_for_layout?></title>
```

```
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
<!-- Inclure les fichiers et scripts externes ici (Voir le HTML Helper pour plus
d'informations) -->
<?php echo $scripts_for_layout ?>
</head>
<body><!-- Si vous aimeriez qu'une sorte de menu se voit sur toutes vos vues,
incluez-le ici -->
<div id="header">
    <div id="menu">...</div>
</div>
<!-- Voici où je veux que mes vues soient affichées -->
<?php echo $content_for_layout ?>

<!-- On ajoute un pied de page à chaque page affichée -->
<div id="footer">...</div>
</body>
</html>
```

`$scripts_for_layout` contient tout fichier externe et script inclus avec le Helper HTML livré avec Cake. Ceci est utile pour inclure des fichiers javascript et CSS depuis les vues.

Lorsque vous utilisez `$html->css()` ou `$javascript->link()` dans les fichiers de vues, spécifiez 'false' pour l'argument 'in-line' afin de placer le code source HTML dans `$scripts_for_layout`. (Voir l'API pour plus de détails sur l'usage).

`$content_for_layout` contient la vue. C'est ici que le code de la vue sera placé.

`$title_for_layout` contient le titre de la page.

Pour définir le titre du gabarit, c'est encore plus simple de le faire depuis le contrôleur, en définissant la variable $title_for_layout.

```
<?php

class UtilisateursController extends AppController {
    function voirActifs() {
        $this->set('title_for_layout', 'Voir les utilisateurs actifs');
    }
}
?>
```

Vous pouvez créer autant de gabarits que vous le souhaitez : placez-les simplement dans le dossier app/views/layouts, et basculez de l'un à l'autre à l'intérieur des actions de vos contrôleurs, en utilisant la variable $layout du contrôleur ou la fonction setLayout().

Par exemple, si une section de mon site incluait un plus petit espace pour les bannières publicitaires, je pourrais créer un nouveau gabarit avec l'espace publicitaire plus petit et le spécifier comme layout pour toutes les actions des contrôleurs en utilisant quelque chose comme :

var $layout = 'petite_pub_defaut';

```
<?php
```

```
class UtilisateursController extends AppController {
    function voirActifs() {
        $this->set('title_for_layout', 'Voir les utilisateurs actifs');
        $this->layout = 'petite_pub_defaut'
    }

    function voirImage() {
        $this->layout = 'image';
        // affiche l'image de l'utilisateur
    }
}
?>
```

CakePHP présente deux gabarits du coeur (en plus du layout par défaut) que vous pouvez utiliser dans vos propres applications : 'ajax' et 'flash'. Le gabarit Ajax est pratique pour façonner les réponses Ajax - c'est un layout vide (la plupart des appels ajax nécessitent seulement un peu de balisage en retour, plutôt qu'une interface entièrement rendue). Le gabarit Flash est utilisé pour l'affichage des messages affichés par la méthode flash() des contrôleurs.

Trois autres gabarits : xml, js et rss - existent dans le coeur afin de générer de manière simple et rapide du contenu qui n'est pas du text/html.

3.10.3 Eléments

De nombreuses applications disposent de petits blocs de présentation qui doivent se répéter de page en page, parfois à des endroits différent du layout. CakePHP peut vous aider à répéter ces parties de votre site web qui ont besoin d'être réutilisées. Ces parties sont appelées Eléments. Avertissements, boîtes d'aide, contrôles de navigation, extra menus, formulaires de login et infos-bulles sont souvent implémentés comme des éléments dans CakePHP. Un élément est simplement une mini-vue qui peut être incluse dans d'autres vues, dans des layouts et même dans d'autres éléments. Les Eléments peuvent être utilisés pour rendre une vue plus lisible, en plaçant les éléments répétitifs dans leur propre fichier. Ils peuvent aussi vous aider à réutiliser des fragments de contenu dans votre application.

Les éléments sont dans le dossier **`/app/views/elements/`** et ont l'extension de fichier .ctp. Ils sont affichés en utilisant la méthode *element* de la vue.

```
<?php echo $this->element('boite_dialogue'); ?>
```

3.10.3.1 Transmettre des variables à un élément

Vous pouvez faire passer des données à un élément *via* le second argument :

```
<?php echo
$this->element('aide',
    array("texteaide" => "Oh, ce texte est vraiment utile."));
?>
```

Dans le fichier correspondant à l'élément, toutes les variables passées sont accessibles dans le tableau *parameter* (de la même manière que, dans le contrôleur, `set()` envoie les variables aux vues). Dans l'exemple ci-dessus, vous pouvez utiliser la variable `$texteaide` dans le fichier **`/app/views/elements/aide.ctp`**.

```
<?php
```

```
echo $texteaide; //affichera "Oh, ce texte est vraiment utile."
?>
```

Dans la fonction `element()` sont regroupées les options de l'élément avec les données à transmettre. Les deux options sont 'cache' et 'plugin'. Par exemple :

```
<?php echo
$this->element('aide',
    array(
        "texteaide" => "Transmis à l'élément dans la variable
$texteaide",
        "toto" => "Transmis à l'élément dans la variable $toto",
        "cache" => "+2 days", // mettra l'élément en cache pour 2 jours.
        "plugin" => "" // pour fournir une partie d'un plugin
    )
);
?>
```

Pour mettre en cache plusieurs versions du même élément dans l'application, vous pouvez attribuer une clé unique de cache de la manière suivante :

```
<?php
$this->element('aide',
    array(
        "cache" => array('time'=> "+7 days",
                         'key'=>'valeur unique')
    )
);
?>
```

Vous pouvez profiter pleinement des élements en utilisant `requestAction()`. Cette fonction retourne dans un tableau les variables de vue à partir d'une action d'un contrôleur. Cela permet à vos éléments de fonctionner en respectant la logique MVC. Créez une action dans un contrôleur qui prépare les variables de vue pour votre élément, et appelez `requestAction()` dans le second paramètre de votre fonction `element()` pour directement passer à l'élément les variables issues du contrôleur.

Pour cela, ajoutez dans votre contrôleur un code similaire au suivant :

```
<?php
class MessagesController extends AppController {
    ...
    function index() {
        $messages = $this->paginate();
        if (isset($this->params['requested'])) {
            return $messages;
        } else {
            $this->set('messages',$messages);
        }
    }
}
?>
```

Puis, dans la vue, nous pouvons accéder aux messages triés par pages. Pour avoir les cinq derniers messages triés, nous pouvons faire quelque chose de ce style :

```
<h2>Derniers messages</h2>
<?php $messages=
$this->requestAction('messages/index/sort:created/direction:asc/limit:5'); ?>
<?php foreach($messages as $message): ?>
<ol>
    <li><?php echo $message['Message']['titre']; ?></li>
</ol>
<?php endforeach; ?>
```

3.10.3.2 Mettre en cache les Éléments

Vous pouvez tirer avantage du cache de vue CakePHP si vous fournissez un paramètre de cache. S'il est défini à true, la mise en cache sera d'un jour. Sinon, vous pouvez définir d'autres délais d'expiration du cache. Voir Mettre en cache pour plus d'information sur les réglages d'expiration.

```
<?php echo $this->element('boite_dialogue', array('cache' => true)); ?>
```

Si vous utilisez le même élément plus d'une fois dans une vue et que vous avez activé la mise en cache, assurez-vous de définir le paramètre 'key' avec un nom différent chaque fois. Cela empêchera chaque appel successif d'écraser le résultat mis en cache lors du précédent appel à element(). Par exemple :

```
<?php
echo $this->element('boite_dialogue', array('cache' => array('key' =>
'premier_usage', 'time' => '+1 day'), 'var' => $variable));

echo $this->element('boite_dialogue', array('cache' => array('key' =>
'second_usage', 'time' => '+1 day'), 'var' => $variableDifferente));
?>
```

De cette façon, on s'assure que chacun des éléments possède sa propre mise en cache, séparée de l'autre.

3.10.3.3 Appeler des Eléments depuis un Plugin

Si vous utilisez un plugin et que vous souhaitez utiliser des éléments avec lui, spécifiez simplement le paramètre plugin. Si la vue doit être rendue pour une action de contrôleur de plugin, elle pointera automatiquement vers l'élément destiné au plugin. Si l'élément n'existe pas dans le plugin, elle regardera dans le dossier **APP** prinicpal.

```
<?php echo $this->element('boite_dialogue', array('plugin' => 'nom_plugin'));
?>
```

3.10.4 Méthodes de Vue

set(string $var, mixed $value)

Les vues ont une méthode `set()` analogue au `set()` trouvé dans les objects Contrôleur. Elle vous permet d'ajouter des variables à viewVars. Utiliser set() depuis votre fichier de vue ajoutera les variables au layout et aux éléments qui seront rendus ultérieurement. Voyez Controller::set() pour plus d'information sur l'utilisation de set().

Dans votre fichier de vue vous pouvez faire

```
    $this->set('activeMenuBouton', 'posts');
```

Ensuite dans votre layout la variable `$activeMenuBouton` sera disponible et contiendra la valeur 'posts'.

3.10.4.2 getVar()

`getVar(string $var)`

Retourne la valeur de viewVar dont le nom est $var

3.10.4.3 getVars()

`getVars()`

Retourne une liste de toutes les variables de vue disponibles dans le champ de rendu courant. Retourne un tableau des noms de variable.

3.10.4.4 error()

`error(int $code, string $name, string $message)`

Affiche une page d'erreur à l'utilisateur. Utilise layouts/error.ctp pour rendre la page.

```
    $this->error(404, 'Non trouvée', 'Cette page n\'a pas été
trouvée, désolé');
```

Ceci rendra une page d'erreur avec le titre et le message spécifiés. Il est important de noter que l'exécution du script n'est pas arrêtée par `View::error()`. Donc vous devrez stopper l'exécution du code vous même, si vous voulez interrompre le script.

3.10.4.5 element()

`element(string $elementPath, array $data, bool $loadHelpers)`

Rends un élément ou une vue partielle. Voyez la section sur View Elements pour plus d'informations et d'exemples.

3.10.4.6 uuid

`uuid(string $object, mixed $url)`

Génère un ID DOM unique non-aléatoire pour un objet, basé sur le type d'objet et l'url. Cette méthode est souvent utilisée par les assistants, tels que AjaxHelper, qui ont besoin de générer des ID DOM uniques pour les éléments.

```
    $uuid = $this->uuid('form', array('controller' => 'posts', 'action' =>
'index'));
    //$uuid contient 'form0425fe3bad'
```

3.10.4.7 addScript()

```
addScript(string $name, string $content)
```

Ajoute du contenu au tampon (*buffer*) de scripts interne. Ce buffer est rendu disponible dans le layout par `$scripts_for_layout`. Cette méthode est pratique pour la création d'assistants qui nécessitent d'ajouter du javascript ou des css directement dans le layout. Gardez à l'esprit que les scripts ajoutés depuis le layout ou depuis les éléments dans le layout ne seront pas ajoutés à `$scripts_for_layout`. Cette méthode est plus souvent utilisée à l'intérieur des assistants, comme dans les *Helpers* Javascript et Html.

3.10.5 Thèmes

Vous pouvez tirer avantage des thèmes pour rendre le changement de look et de design de vos pages plus rapide et plus aisé.

Pour utiliser les thèmes, vous devez spécifier à votre contrôleur d'utiliser la classe *ThemeView* plutôt que la classe par défaut *View*.

```
class ExemplesController extends AppController {
    var $view = 'Theme';
}
```

Pour déclarer quel thème utiliser par défaut, spécifiez le nom de ce thème dans votre contrôleur :

```
class ExemplesController extends AppController {
    var $view = 'Theme';
    var $theme = 'exemple_theme';
}
```

Vous pouvez également choisir ou changer le thème dans une action, ou dans les fonctions de rappel `beforeFilter` ou `beforeRender`.

```
$this->theme = 'autre_exemple_theme';
```

Les vues de votre thème doivent être situées dans le dossier /app/views/themed/. Dans ce dossier, créez un nouveau dossier du même nom que votre thème, par exemple /app/views/themed/exemple_theme/. Ensuite, sa structure sera exactement la même que celle de /app/views/.

Par exemple, la vue de votre fonction `editer()` d'un contrôleur Messages se situerait à /app/views/themed/exemple_theme/messages/editer.ctp. Les fichiers de mise en page (*Layout*) seraient quant à eux dans le dossier /app/views/themed/exemple_theme/layouts/.

Si CakePHP ne peut trouver la vue dans le dossier du thème, il la cherchera dans le dossier /app/views/. De cette manière, vous pouvez créer des vues générales, et simplement les adapter à votre thème, au cas-par-cas, dans le dossier adapté.

Si vous avez des fichiers CSS ou JavaScript qui sont spécifiques à votre thème, vous pouvez les stocker dans un dossier de thème à l'intérieur de webroot. Vos feuilles de style seront par exemple situées dans /app/webroot/themed/exemple_theme/css/, et vos fichiers JavaScript seront dans /app/webroot/themed/exemple_theme/js/.

Tous les Helpers natifs de CakePHP savent gérer les thèmes, et créeront les bons chemins d'accès de façon automatique. De même qu'avec les vues, si un fichier n'est pas dans le dossier du thème, l'utilisateur sera renvoyé par défaut à la racine.

3.10.5.1 Increasing performance of plugin and theme assets

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Its a well known fact that serving assets through PHP is guaranteed to be slower than serving those assets without invoking PHP. And while the core team has taken steps to make plugin and theme asset serving as fast as possible, there may be situations where more performance is required. In these situations its recommended that you either symlink or copy out plugin/theme assets to directories in `app/webroot` with paths matching those used by cakephp.

becomes `app/webroot/debug_kit/js/my_file.js` `app/webroot/theme/navy/css/navy.css`

3.10.6 Vues Media

Les vues de media vous permettent d'envoyer des fichiers binaires à l'utilisateur. Par exemple, vous pouvez souhaiter avoir un dossier de fichiers extérieur à votre webroot afin d'empêcher les utilisateurs d'avoir un lien direct vers eux. Vous pouvez utiliser la vue de Media pour récupérer le fichier depuis un dossier spécial de /app/, ce qui vous permet d'effectuer une authentification avant de délivrer le fichier à l'utilisateur.

Pour utiliser la vue Media, vous devez dire à vôtre contrôleur d'utiliser la classe MediaView au lieu de la classe par défaut View. Après ça, passez simplement des paramètres additionnels pour spécifier l'emplacement de votre fichier.

```
class ExempleController extends AppController {
    function telecharger () {
        $this->view = 'Media';
        $params = array(
              'id' => 'exemple.zip',
              'name' => 'exemple',
              'download' => true,
              'extension' => 'zip',
              'path' => 'fichiers' . DS
       );
       $this->set($params);
    }
}
```

Voici un exemple de rendu d'un fichier dont le type mime n'est pas inclut dans le tableau `$mimeType` de MediaView.

```
function telecharger () {
    $this->view = 'Media';
    $params = array(
        'id' => 'exemple.docx',
        'name' => 'exemple',
        'extension' => 'docx',
        'mimeType' => array('docx' =>
'application/vnd.openxmlformats-officedocument.wordprocessingml.document'),
        'path' => APP . 'fichiers' . DS
    );
    $this->set($params);
```

```
}
```

Paramètres	Description
id	L'ID est le nom du fichier tel qu'il est sur le serveur de fichiers, extension incluse.
name	Le nom vous permet de spécifier un nom de fichier alternatif qui sera envoyé à l'utilisateur. Spécifiez le nom sans l'extension du fichier.
download	Une valeur booléenne indiquant si les en-têtes doivent être définis pour forcer le téléchargement. Notez que l'option autoRender de votre contrôleur doit être définie à `false` pour que ceci fonctionne correctement.
extension	L'extension du fichier. Ceci est comparé avec une liste interne de types mime acceptables. Si le type mime spécifié n'est pas dans la liste, le fichier ne sera pas téléchargé.
path	Le nom du dossier, incluant le séparateur de dossiers final. Le chemin devrait être absolu, mais il peut être relatif au dossier APP/webroot.
mimeType	Un tableau avec des types mime additionnels qui seront fusionnés avec la liste interne des types mime acceptables présent dans la classe MediaView.
cache	Une valeur booléenne ou entière - Si définie à *true*, autorise les navigateurs à mettre le fichier en cache (si non définie, défaut à *false*) ; autrement, définissez-la à un nombre de secondes à partir duquel le cache doit expirer.

3.11 Assistants

Vous utilisez les assistants (*helpers*) dans CakePHP, en faisant "prendre conscience" à un contrôleur qu'ils existent. Chaque contrôleur a une propriété $helpers, qui liste les assistants disponibles dans la vue. Pour activer un assistant dans votre vue, ajoutez son nom au tableau $helpers du contrôleur.

```
<?php
class BoulangeriesController extends AppController {
    var $helpers = array('Form', 'Html', 'Javascript', 'Time');
}
?>
```

Vous pouvez aussi ajoutez les assistants depuis une action, dans ce cas, ils seront uniquement accessibles pour cette action et aucune autre dans le contrôleur. Ceci économise de la puissance de calcul pour les autres actions qui n'utilisent pas l'assistant, tout en permettant de conserver le contrôleur mieux organisé.

```
<?php
class BoulangeriesController extends AppController {
    function cuire {
        $this->helpers[] = 'Time';
    }
    function melange {
        // L'assistant Time n'est pas chargé ici et par conséquent non
disponible
    }
}
?>
```

Si vous avez besoin d'activer un assistant pour tous les contrôleurs, ajoutez son nom dans le tableau `$helpers` du fichier */app/app_controller.php* (à créer si pas présent). Souvenez-vous d'inclure les assistants par défaut Html et Form.

```
<?php
class AppController extends Controller {
    var $helpers = array('Form', 'Html', 'Javascript', 'Time');
}
?>
```

3.11.2 Créer des Assistants

Si un assistant du coeur (ou l'un de ceux présentés sur Cakeforge ou dans la Boulangerie) ne répond pas à vos attentes, il est facile de créer des assistants.

Mettons que nous voulions créer un assistant, 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 d'assistant existante dans CakePHP, vous devrez créer une nouvelle classe dans /app/views/helpers. Appelons notre assistant LienHelper. Le fichier de la classe PHP devrait ressembler à quelque chose comme ceci :

```
<?php
/* /app/views/helpers/lien.php */

class LienHelper extends AppHelper {
    function lancerEdition($titre, $url) {
        // La logique pour créer le lien spécialement formaté se
place ici...
    }
}

?>
```

Il y a quelques méthodes incluent dans la classe Helper de CakePHP, dont vous pourriez tirer avantage :

`output(string $string)`

Utilisez cette fonction pour transmettre toutes données à votre vue.

```
<?php
function lancerEdition($titre, $url) {
    // Utilisez la fonction output de la classe Helper
    // pour transmettre des données formatées à votre vue :
    return $this->output(
        "<div class=\"contourEdition\">
         <a href=\"$url\" class=\"editer\">$titre</a>
         </div>"
    );
}
?>
```

3.11.2.1 Inclure d'autres Assistants

Vous souhaitez peut-être utiliser quelques fonctionnalités déjà existantes dans un autre assistant. Pour faire cela, vous pouvez spécifier les assistants que vous souhaitez utiliser avec un tableau $helpers, formaté comme vous le feriez dans un contrôleur.

```
<?php
/* /app/views/helpers/lien.php (utilisant d'autres assistants) */
class LienHelper extends AppHelper {
    var $helpers = array('Html');

    function lancerEdition($titre, $url) {
        // Utilisez l'assistant HTML pour transmettre
        // les données formatées :

        $lien = $this->Html->link($titre, $url, array('class' => 'editer'));

        return $this->output("<div class=\"contourEdition\">$lien</div>");
    }
}
?>
```

3.11.2.2 Méthode de Rappel (callback)

Les Assistants présentent un *callback* utilisé par la classe contrôleur parente.

`beforeRender()`

La méthode beforeRender est appelée après la méthode beforeRender du contrôleur, mais avant le rendu des vues et du gabarit.

3.11.2.3 Utiliser votre Assistant

Une fois que vous avez créé votre assistant et que vous l'avez placé dans /app/views/helpers/, vous serez en mesure de l'inclure dans vos contrôleurs, en utilisant la variable spéciale $helpers.

Une fois que votre contrôleur a été informé de cette nouvelle classe, vous pouvez l'utiliser dans vos vues, en accédant à une variable nommée d'après le nom de l'assistant :

```
<!-- créer un lien en utilisant le nouvel assistant -->
<?php echo $lien->lancerEdition('Changer cette Recette', '/recettes/editer/5') ?>
```

Les helpers Html, Form et Session (si les sessions sont activées) sont toujours accessibles.

3.11.3 Créer des Fonctionnalités pour Tous les Assistants

Tous les assistants étendent une classe spéciale, AppHelper (tout comme les modèles étendent AppModel et les contrôleurs étendent AppController). Pour créer une fonctionnalité qui devrait être disponible pour tous les assistants, créez **`/app/app_helper.php`**.

```
<?php
class AppHelper extends Helper {
    function methodePerso () {
    }
```

```
}
?>
```

3.11.4 Assistants intégrés

CakePHP fournit bon nombre d'assistants (*helpers*) qui vous aideront dans la création de vues. Ils vous assistent à la création de marquage bien formaté (dont les formulaires), vous aident à formater du texte, des heures et des nombres, et peuvent même accélérer les fonctionnalités Ajax. Voici un résumé de assistants livrés de base. Pour plus d'informations, rendez-vous sur Assistants intégrés.

Assistant CakePHP	Description
Ajax	Utilisé en tandem avec la librairie javascript Prototype pour créer des fonctionnalités Ajax dans les vues. Il contient des méthodes de raccourci pour le drag&drop, les formulaires et liens Ajax, les observateurs, et bien plus.
Cache	Utilisé par le coeur pour mettre en cache le contenu des vues.
Form	Crée des formulaires HTML et des éléments de formulaire qui se remplissent automatiquement avec les bonnes valeurs et qui gèrent les problèmes de validation.
Html	Des méthodes pratiques pour mettre en oeuvre un marquage bien formé. Des images, liens, tableaux, balises d'en-tête et plus encore.
Javascript	Utilisé pour échapper des valeurs utilisées dans des scripts Javascripts, écrire des données en objets JSON, et formater des blocs de code.
Number	Formatage de nombres et monnaies.
Paginator	Pagination et tri des données de modèles.
Rss	Des méthodes pratiques pour produire des données XML de flux RSS.
Session	Accès pour l'écriture en session de valeurs depuis les vues.
Text	Liens élégants, surbrillance, troncature de mot judicieuse.
Time	Détection de proximité (est-ce l'an prochain ?), formatage de belles chaînes de caractères (Aujourd'hui, 10h30) et conversion de fuseaux horaires.
Xml	Des méthodes pratiques pour créer des entêtes et éléments XML.

3.12 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 modèle et de son contrôleur. Déclarez la variable `$scaffold` dans le contrôleur, 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 contrôleur :

```
<?php

class CategoriesController extends AppController {
    var $scaffold;
}

?>
```

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

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

Le *Scaffolding* prend bien en compte les relations contenues dans votre modèle. Ainsi, si votre modèle Category a une relation BelongsTo avec le modèle Utilisateur, vous verrez les identifiants des utilisateurs dans l'affichage de vos catégories. Si vous préférez voir autre chose en plus des identifiants (par exemple les prénoms des utilisateurs), vous pouvez affecter la variable `$displayField` dans le modèle.

Voyons comme définir la variable `$displayField` dans la classe des utilisateurs, 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.

```
<?php

class Utilisateur extends AppModel {
    var $name = 'Utilisateur';
    var $displayField = 'prenom';
}

?>
```

3.12.1 Créer une interface d'administration simple avec le 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*.

```
var $scaffold = 'admin';
```

Vous serez maintenant capable d'accéder aux actions *scaffoldées* :

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

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

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

3.12.2 Modifier 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 peut être obtenue en créant des *templates*.

```
Les vues de scaffolding personnalisées pour un contrôleur spécifique (MessagesController dans

/app/views/messages/scaffold.index.ctp
/app/views/messages/scaffold.show.ctp
/app/views/messages/scaffold.edit.ctp
/app/views/messages/scaffold.new.ctp

Les vues de scaffolding personnalisées pour tous les contrôleurs doivent être agencées comme c

/app/views/scaffolds/index.ctp
/app/views/scaffolds/show.ctp
/app/views/scaffolds/edit.ctp
/app/views/scaffolds/new.ctp
/app/views/scaffolds/add.ctp
```

3.13 La console CakePHP

Cette section est une introduction dans CakePHP depuis la ligne de commande. Si vous avez déjà eu besoin d'accéder à vos classes MVC CakePHP depuis une tâche cron ou tout autre script en 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.

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 des spécificités, soyons sÃ»rs 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.

Vous pouvez techniquement lancer la console en utilisant quelquechose comme :

```
$ cd /my/cake/app_folder
$ ../cake/console/cake
```

Mais il est préférable d'ajouter le dossier de la console à votre path afin que vous puissiez utiliser la commande cake de partout :

```
$ cake
```

Exécuter la Console sans arguments produit ce message d'aide (en anglais à l'heure actuelle) :

```
Hello user,

Welcome to CakePHP v1.2 Console
---------------------------------------------------------------
Current Paths:
 -working: /path/to/cake/
 -root: /path/to/cake/
 -app: /path/to/cake/app/
 -core: /path/to/cake/

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/myapp or -app /absolute/path/to/myapp

Available Shells:

 app/vendors/shells/:
         - none
```

```
 vendors/shells/:
        - none

 cake/console/libs/:
        acl
        api
        bake
        console
        extract

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 /chemin/vers/app
```

Le chemin fourni peut être relatif au répertoire courant ou fourni sous forme de chemin absolu.

3.13.1 Créer des Shells & des Tâches

Créons un shell qui sera utilisé dans la Console. Dans cet exemple, nous allons créer un shell de "rapport" qui affiche quelques données du modèle. D'abord, créons rapport.php dans /vendors/shells/.

```
<?php
class RapportShell extends Shell {
    function main() {}
}
?>
```

Dès à présent, nous pouvons lancer le shell, mais cela ne fera pas grand chose. Ajoutons quelques modèles au shell afin de pouvoir créer un rapport sur quelque chose. Ceci se fait de la même manière que dans le contrôleur : en ajoutant le nom des modèles à utiliser à la variable $uses.

```
<?php
class RapportShell extends Shell {
    var $uses = array('Commande');

    function main() {
    }
}
?>
```

Une fois que nous avons ajouté notre modèle au tableau $uses, nous pouvons l'utiliser dans la méthode main(). Dans cet exemple, notre modèle Commande devrait désormais être accessible par $this->Commande dans la

méthode main() de notre nouveau shell.

Voici un exemple simple de la logique que nous pourrions utiliser dans ce shell :

```
class RapportShell extends Shell {
    var $uses = array('Commande');
    function main() {
        // Récupérer les commandes livrées le mois dernier
        $mois_dernier = date('Y-m-d H:i:s', strtotime('-1 month'));
        $achats = $this->Commande->find("all",array('conditions'=>"Commande.expedition
>= '$mois_dernier'"));

        // Affiche les informations de chaque commande
        foreach($achats as $achat) {
            $this->out('Date de la commande : '.$achat['Commande']['created']."\n");
            $this->out('Montant : $'.number_format($achat['Commande']['montant'], 2)."\n");
            $this->out('----------------------------------------'."\n");

            $total += $achat['Commande']['montant'];
        }

        // Affiche le total des commandes sélectionnées
        $this->out("Total : $".number_format($total, 2)."\n");
    }
}
```

Vous devriez pouvoir lancer ce rapport en exécutant cette commande (si la commande cake est dans votre PATH) :

```
$ cake rapport
```

où rapport est le nom de fichier du shell dans /vendor/shells/ sans l'extension .php. Cela devrait produire quelque chose comme :

```
Hello user,
   Welcome to    CakePHP v1.2 Console
   ---------------------------------------------------------------
   App : app
   Path:    /path/to/cake/app
   ---------------------------------------------------------------
   Date de la commande :    2007-07-30 10:31:12
   Montant :    $42.78
   ----------------------------------------
   Date de la commande :     2007-07-30 21:16:03
   Montant :    $83.63
   ----------------------------------------
   Date de la commandet :    2007-07-29 15:52:42
   Montant :    $423.26
   ----------------------------------------
   Date de la commande :    2007-07-29 01:42:22
   Montant :    $134.52
   ----------------------------------------
   Date de la commande :    2007-07-29 01:40:52
   Montant :    $183.56
   ----------------------------------------
   Total:    $867.75
```

3.13.1.2 Tâches

Les Tâches sont des petites extensions des shells. Elles permettent de partager de la logique entre des shells, et sont ajoutées aux shells en utilisant la variable de classe spéciale $tasks. Par exemple dans le shell du coeur (*bake*), il y a un certain nombre de tâches définies :

```
<?php
class BakeShell extends Shell {
   var $tasks = array('Project', 'DbConfig', 'Model', 'View', 'Controller');
}
?>
```

Les tâches sont stockées dans /vendors/shells/tasks/ dans des fichiers aux noms de leurs classes. Si nous voulions créer une tâche "cool", la classe CoolTask (qui *extends Shell*) sera placée dans /vendors/shells/tasks/cool.php.

Chaque tâche doit au moins implémenter la méthode execute() - les shells appelleront cette méthode pour démarrer la logique de la tâche.

```
<?php
class SonTask extends Shell {
   var $uses = array('Model'); // identique à la variable de contrôleur $uses
   function execute() {}
}
?>
```

Vous pouvez accéder à la tâche depuis vos classes de shell et les exécuter là-bas :

```
<?php
class MerShell extends Shell // dans /vendors/shells/mer.php {
   var $tasks = array('Son'); // dans /vendors/shells/tasks/son.php
   function main() {
       $this->Son->execute();
   }
}
?>
```

Une méthode appelée "son" dans la classe MerShell aurait surchargé la possibilité d'accéder à la fonctionnalité de la tâche Son spécifiée dans le tableau $tasks.

Vous pouvez aussi accéder aux tâches directement depuis la ligne de commande :

```
$ cake mer son
```

3.13.2 Exécuter des Shells en tâches cron

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. Cependant, même si vous avez ajouté le chemin de la console à la variable PATH via `~/.profile`, elle sera indisponible pour la tâche cron.

Le script BASH suivant appellera votre shell et ajoutera les chemins nécessaires à $PATH. Copiez et sauvegardez ceci dans votre dossier vendors, en le nommant 'cakeshell' et n'oubliez pas de le rendre exécutable. (`chmod +x cakeshell`)

```
#!/bin/bash
TERM=dumb
export TERM
cmd="cake"
while [ $# -ne 0 ]; do
        if [ "$1" = "-cli" ] || [ "$1" = "-console" ]; then
                PATH=$PATH:$2
                shift
        else
                cmd="${cmd} $1"
        fi
        shift
done
$cmd
```

Vous pouvez l'appeler comme suit :

```
$ ./vendors/cakeshell monshell monparam -cli /usr/bin -console /cakes/1.2.x.x/cake/console
```

Le paramètre `-cli` prend un chemin qui pointe vers l'exécutable cli php et le paramètre `-console` prend un chemin qui pointe vers la console CakePHP.

Pour une tâche cron, ceci devrait ressembler à :

```
# m h  dom mon dow   command
*/5 * * * * /chemin/complet/vers/cakeshell monshell monparam -cli /usr/bin -console /cakes/1.2
```

Un truc simple pour débugger une *crontab*, c'est de la paramétrer pour qu'elle vide sa sortie dans un fichier de log. Vous pouvez faire ceci comme çà :

```
# m h  dom mon dow   command
*/5 * * * * /chemin/complet/vers/cakeshell monshell monparam -cli /usr/bin -console /cakes/1.2
```

3.14 Plugins

CakePHP vous permet de mettre en place une combinaison de contrôleurs, modèles 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.

3.14.1 Créer un Plugin

Comme exemple de travail, créons un nouveau plugin qui commande des pizzas pour vous. Pour commencer,

nous aurons besoin de placer les fichiers de notre plugin dans le dossier /app/plugins. Le nom du dossier parent pour tous les fichiers de plugin est important, et sera utilisé dans plusieurs endroits, alors choisissez-le judicieusement. Pour ce plugin, utilisons le nom '**pizza**'. Voici comme votre installation devrait éventuellement se présenter :

```
/app
    /plugins
        /pizza
            /controllers                <- les contrôleurs du plugin vont ici
            /models                     <- les modèles du plugin vont ici
            /views                      <- les vues du plugin vont ici
            /pizza_app_controller.php   <- AppController du plugin
            /pizza_app_model.php        <- AppModel du plugin
```

Si vous voulez pouvoir accéder à votre plugin par une URL, il est obligatoire de définir AppController et AppModel pour ce plugin. Ces deux classes spéciales se nomment de la même manière que le plugin et étendent du AppController et AppModel de l'application parente. Voici ce à quoi ils devraient ressembler dans notre exemple de pizza :

```
// /app/plugins/pizza/pizza_app_controller.php:
<?php
class PizzaAppController extends AppController {
    //...
}
?>
```

```
// /app/plugins/pizza/pizza_app_model.php:
<?php
class PizzaAppModel extends AppModel {
      //...
}
?>
```

Si vous oubliez de définir ces classes spéciales, CakePHP vous délivrera une erreur "Contrôleur manquant" ("*Missing Controller*") jusqu'à ce que vous l'ayez fait.

3.14.2 Contrôleurs du Plugin

Les contrôleurs de notre plugin de pizza seront stockés dans /app/plugins/pizza/controllers/. Comme la principale chose que nous allons pister sont des commandes de pizza, nous aurons besoin d'un contrôleur CommandesController pour ce plugin.

Bien que cela ne soit pas obligatoire, il est recommandé de nommer les contrôleurs de vos Plugins d'une manière relativement unique afin d'éviter des conflits d'espaces de noms avec les applications parentes. Il n'est pas idiot de penser qu'une application parente peut avoir un contrôleur UtilisateursController, CommandesController, ou ProduitsController : c'est pour cela qu'il vous faudra être créatif avec les noms de vos contrôleurs, ou bien ajouter le nom du plugin au nom de la classe (PizzaCommandesController dans ce cas).

Donc, nous plaçons notre nouveau PizzaCommandesController dans /app/plugins/pizza/controllers et il ressemble à ceci :

```
// /app/plugins/pizza/controllers/pizza_commandes_controller.php
class PizzaCommandesController extends PizzaAppController {
```

```
    var $name = 'PizzaCommandes';
    var $uses = array('Pizza.PizzaCommande');
    function index() {
          //...
    }
}
```

Ce contrôleur hérite du AppController du plugin (appelé PizzaAppController) plutôt que du AppController de l'application parente.

Notons également que le nom du modèle est préfixé avec le nom du plugin. Cette ligne de code est ajoutée pour être plus clair mais n'est pas nécessaire pour cet exemple.

Si vous voulez accéder à ce que nous avons fait jusqu'à présent, visitez /pizza/pizza_commandes. Vous devriez obtenir une erreur "Modèle manquant" ("Missing Model") car nous n'avons pas encore de modèle PizzaCommande défini.

3.14.3 Modèles du Plugin

Les modèles de notre plugin de pizza seront stockés dans /app/plugins/pizza/models/. Nous avons déjà défini un contrôleur PizzaCommandesController pour ce plugin, alors créons le modèle de ce contrôleur, nommé PizzaCommande. PizzaCommande est cohérent avec notre schéma de nommage précédemment défini, à savoir préfixer toutes nos classes de plugin avec Pizza.

```
// /app/plugins/pizza/models/pizza_commande.php:
class PizzaCommande extends PizzaAppModel {
    var $name = 'PizzaCommande';
}
?>
```

Visiter /pizza/pizzaCommandes maintenant (en supposant que vous ayez une table nommée "pizza_commandes" dans votre base de données) devrait vous donner une erreur "Vue manquante" ("*Missing View*"). Nous la créerons après.

Si vous avez besoin de référencer un modèle dans votre plugin, vous devrez inclure le nom du plugin avec le nom du modèle, séparés par un point.

Par exemple :

```
// /app/plugins/pizza/models/pizza_commande.php:
class ModeleExemple extends PizzaAppModel {
    var $name = 'ModeleExemple';
        var $hasMany = array('Pizza.PizzaCommande');
}
?>
```

Si vous préférez que les clés du tableau de l'association n'aient pas le préfixe du plugin avec elles, utilisez la syntaxe alternative :

```
// /app/plugins/pizza/models/pizza_commande.php:
class ModeleExemple extends PizzaAppModel {
    var $name = 'ModeleExemple';
        var $hasMany = array(
                'PizzaCommande' => array(
                        'className' => 'Pizza.PizzaCommande'
                )
        );
}
?>
```

3.14.4 Vues du plugin

Les vues se comportent exactement comme elles le font dans les applications normales. Placez les simplement dans le bon dossier à l'intérieur du dossier /app/plugins/[plugin]/views/ Pour notre plugin de commande de pizza, nous aurons besoin d'une vue pour notre action PizzaCommandesController::index(), alors ajoutons là également :

```
// /app/plugins/pizza/views/pizza_commandes/index.ctp:
<h1>Commander une pizza</h1>
<p>Rien ne va mieux avec un Gâteau qu'une bonne pizza !</p>
<!-- Un formulaire de commande devrait être ici ...-->
```

Pour plus d'information sur l'utilisation d'éléments à partir d'un plugin, allez voir Appeler des éléments à partir d'un plugin

Overriding plugin views from inside your application

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

You can override any plugin views from inside your app using special paths. If you have a plugin called 'Pizza' you can override the view files of the plugin with more application specific view logic by creating files using the following template "app/views/plugins/$plugin/$controller/$view.ctp". For the pizza controller you could make the following file:

```
/app/views/plugins/pizza/pizza_orders/index.ctp
```

Creating this file, would allow you to override "/app/plugins/pizza/views/pizza_orders/index.ctp".

3.14.5 Composants, Assistants et Comportements

Un plugin peut avoir des Composants, des Assistants et des Comportements comme toute application CakePHP normale. Vous pouvez même créer des plugins qui ne soient que des Composants, Assistants ou Comportements, ce qui peut être une belle manière de construire des composants réutilisables, qui pourront être ajoutés facilement à tout projet.

Construire ces composants est exactement la même chose que de les construire dans une application classique, sans convention particulière de nommage. Se référer à vos composants à l'intérieur du plugin, n'exige pas non plus de référence particulière.

```
// Composant
class ExempleComponent extends Object {

}

// dans le contrôleur de votre Plugin
var $components = array('Exemple');
```

Pour référencer le Composant à l'extérieur du plugin, il est nécessaire de référencer le nom du plugin.

```
var $components = array('PluginNom.Exemple');
var $components = array('Pizza.Exemple'); // référence ExempleComponent du
plugin Pizza.
```

La même technique est appliquée aux Assistants et Comportements.

3.14.6 Images, CSS et Javascript de Plugin

Vous pouvez ajouter des fichiers Images, Javascripts ou CSS spécifiques dans vos plugins. Ces fichiers associés doivent être respectivement placés dans `votre_plugin/vendors/img`, `votre_plugin/vendors/css` et `votre_plugin/vendors/js`. Ils peuvent être liés à vos vues tout comme les helpers du core.

```
<?php echo $html->image('/votre_plugin/img/mon_image.png'); ?>

<?php echo $html->css('/votre_plugin/css/mon_css'); ?>

<?php echo $javascript->link('/votre_plugin/js/super_script');
```

Les exemples ci-dessus montrent comment lier des fichiers images, javascript et CSS à votre plugin.

Il est important de noter le préfixe **/votre_plugin/** devant le chemin des img, js ou css. C'est ce qui fait que ça marche !

3.14.7 Conseils et astuces

Bien, maintenant que vous avez tout développé, vous êtes prêt à distribuer votre plugin (nous vous suggérons d'ailleurs d'y joindre quelques bonus, comme un fichier LisezMoi ou un fichier SQL).

Une fois que le plugin a été installé dans /app/plugins, vous pouvez y accéder à partir de l'url /nomduplugin/nomducontroleur/action. Dans notre exemple de commandes de pizzas, nous accéderons à notre CommandePizzaController par l'url /pizza/commandePizzas.

Encore quelques conseils et astuces pour travailler avec des plugins dans vos applications CakePHP :

- Quand vous n'avez pas un [plugin]AppController et un [plugin]AppModel, vous aurez un message d'erreur "contrôleur manquant" quand vous tenterez d'accéder au contrôleur du plugin.
- Vous pouvez avoir un contrôleur par défaut avec le nom de votre plugin. Si vous faites cela, vous pouvez y accéder par /[plugin]/action. Par exemple, un plugin "utilisateurs" avec un contrôleur "UtilisateursController" peut être appelé avec /utilisateurs/ajout s'il n'y a pas de contrôleur appelé AjoutController dans votre dossier [plugin]/controllers.

- Vous pouvez définir vos propres mises en page pour vos plugins, dans le dossier /app/plugins/[plugin]/views/layouts. Si ce n'est pas le cas, les plugins utiliseront par défaut les mises en page du dossier /app/views/layouts.
- Vous pouvez faire communiquer vos plugins entre eux en utilisant $this->requestAction('/plugin/controller/action') dans vos contrôleurs.
- Si vous utilisez *requestAction*, soyez certain que vos contrôleurs et vos modèles aient des noms uniques. Dans le cas contraire, vous risquez d'avoir des erreurs PHP "classe redéfinie...".

3.15 Constantes et fonctions globales

Tandis que la plupart de votre travail avec CakePHP se basera sur les classes et les méthodes du *Core*, CakePHP dispose d'un certain nombre de fonctions utiles qui seront à votre portée de main pour faciliter votre travail. La plupart de ces fonctions sont utilisées avec les classes CakePHP (en chargeant un modèle ou une classe de composants), et les autres vous faciliteront le travail avec les tableaux et les chaînes de caractères.

Nous allons aussi couvrir la plupart des constantes contenues dans les applications CakePHP. Utiliser ces constantes vous permettra de rendre vos mises-à-jour plus rapides, mais sont également un moyen pratique de pointer vers certains fichiers ou répertoires de votre application CakePHP.

3.15.1 Fonctions globales

__(string $string_id, boolean $return = false)

Cette fonction s'occupe de la localisation dans les applications CakePHP. Le paramètre `$string_id` identifie l'ID pour une traduction et le second paramètre vous permet de déterminer si la fonction affiche automatiquement la chaîne (c'est le comportement par défaut) ou si elle la retourne pour effectuer d'autres traitements (passer un booléen true pour activer ce comportement).

Lisez la section Localisation & Internationalisation pour plus d'information.

3.15.1.2 a

a(mixed $one, $two, $three...)

Retourne un tableau des paramètres utilisés pour appeler la fonction.

```
print_r(a('un', 'deux'));

// affiche :
array(
    [0] => 'un',
    [1] => 'deux'
)
```

3.15.1.3 aa

aa(array $un, $deux, $trois...)

Utilisez pour créer des tableaux associatifs formés par les paramètres utilisés pour appeler la fonction.

Utilisé pour créer des tableaux associatifs formés de paramètres utilisés pour appeler la fonction enveloppante.

```php
echo aa('a','b');

// affiche :
array(
    'a' => 'b'
)
```

3.15.1.4 am

`am(array $one, $two, $three...)`

Fusionne tous les tableaux passés comme paramètres et retourne le tableau résultant.

3.15.1.5 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. Prends un nombre optionnel d'arguments.

Exemple : `config('un_fichier', 'maconfig');`

3.15.1.6 convertSlash

`convertSlash(string $string)`

Convertit les *slashes* en *underscores* et supprime le premier et le dernier *underscores* dans une chaîne. Retourne la chaîne convertie.

3.15.1.7 debug

`debug(mixed $var, boolean $showHtml = false)`

Si le niveau de DEBUG de l'application est différent de zéro, `$var` est affiché. Si `$showHTML` est vrai, la donnée est formatée pour être visualisée facilement dans un navigateur.

3.15.1.8 e

`e(mixed $data)`

Raccourci pratique pour `echo()`.

3.15.1.9 env

`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.

3.15.1.10 fileExistsInPath

```
fileExistsInPath(string $fichier)
```

Vérifie que le fichier donné est dans le include_path PHP actuel. Renvoie une valeur booléenne.

3.15.1.11 h

```
h(string $texte, string $charset = null)
```

Raccourci pratique pour `htmlspecialchars()`.

3.15.1.12 ife

```
ife($condition, $siNonVide, $siVide)
```

Utilisé pour des opérations de styles ternaires. Si `$condition` n'est pas vide, `$siNonVide` est retourné, sinon `$siVide` est retourné.

3.15.1.13 low

```
low(string $chaine)
```

Raccourci pratique pour `strtolower()`.

3.15.1.14 pr

```
pr(mixed $var)
```

Raccourci pratique pour `print_r()`, avec un ajout de balises <pre> autour du résultat (sortie).

3.15.1.15 r

```
r(string $recherche, string $remplace, string $sujet)
```

Raccourci pratique pour `str_replace()`.

3.15.1.16 stripslashes_deep

```
stripslashes_deep(array $valeur)
```

Enlève récursivement les slashes de la `$valeur` passée. Renvoie le tableau modifié.

3.15.1.17 up

```
up(string $chaine)
```

Raccourci pratique pour `strtoupper()`.

3.15.1.18 uses

```
uses(string $lib1, $lib2, $lib3...)
```

Utilisé pour charger les librairies du coeur de CakePHP (trouvables dans **`cake/libs/`**). Passer le nom du fichier de librairie sans l'extension '**`.php`**'.

3.15.2 Définition des Constantes du Coeur

Constante	Chemin absolu vers les éléments suivants :
APP	répertoire racine.
APP_PATH	répertoire de l'application.
CACHE	répertoire des fichiers de cache.
CAKE	répertoire cake.
COMPONENTS	répertoire des composants (*components*).
CONFIGS	répertoire des fichiers de configuration.
CONTROLLER_TESTS	répertoire des tests de contrôleurs.
CONTROLLERS	répertoires des contrôleurs.
CSS	répertoire des fichiers CSS.
DS	Raccourci pour la constante PHP DIRECTORY_SEPARATOR, qui est égale à "/" pour Linux et "\" pour Windows.
ELEMENTS	répertoire des éléments.
HELPER_TESTS	répertoire des tests d'assistant (*helper*).
HELPERS	répertoire des assistants (*helpers*).
IMAGES	répertoire des images.
INFLECTIONS	répertoire des inflexions (habituellement à l'intérieur du répertoire de configuration).
JS	répertoire des fichiers JavaScript (dans le *webroot*).
LAYOUTS	répertoire des mises en pages (*layouts*).
LIB_TESTS	répertoire des tests de la Librairie CakePHP.
LIBS	répertoire des librairies de CakePHP.
LOGS	répertoire des logs (dans app).
MODEL_TESTS	répertoire des tests de modèle.
MODELS	répertoire des modèles.
SCRIPTS	répertoire des scripts Cake.
TESTS	répertoire des tests (répertoire parent des répertoires test des modèles, contrôleurs, etc.)
TMP	répertoire tmp.
VENDORS	répertoire *vendors*.
VIEWS	répertoire des vues.
WWW_ROOT	chemin absolu vers le *webroot*.

3.16 Vendor packages

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Support for vendor assets have been removed for 1.3. It is recommended that you take any vendor assets you have and repackage them into plugins. See Plugin assets for more information.

4 Tâches courantes avec CakePHP

Cette section vous apportera des explications au sujet des tâches courantes suivantes :

- Validation des données
- Sécurisation des données
- Gestion des erreurs
- Débogage
- Mise en cache
- Enregistrement (logging)
- Tester
- Localisation & Internationalisation
- Pagination
- REST

4.1 Validation des données

La validation des données est une partie importante de toute application, puisqu'elle permet de s'assurer que les données d'un modèle 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'utilisateurs 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 modèle des choses. En résumé : ce qui se produit lorsque vous appelez la méthode save() de votre modèle. Pour obtenir plus d'informations sur la manière d'afficher les erreurs de validation, regardez la section traitant de l'assistant Form.

La première étape pour la validation de données est de créer les règles dans le Modèle. Pour ce faire, utilisez le tableau Model::validate dans la définition du modèle, par exemple :

```
<?php
class Utilisateur extends AppModel {
    var $name = 'Utilisateur';
    var $validate = array();
}
?>
```

Dans l'exemple ci-dessus, le tableau $validate est ajouté au modèle Utilisateur, mais ce tableau ne contient pas de règles de validation. En supposant que la table "utilisateurs" ait les champs "login", "mot_de_passe", "email" et "date_de_naissance", l'exemple ci-dessous montre quelques règles simples de validation qui s'appliquent à ces champs :

```
<?php
class Utilisateur extends AppModel {
    var $name = 'Utilisateur';
    var $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 modèle. 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 avantage de quelques-unes de ces règles pré-définies :

```
<?php
class User extends AppModel {
    var $name = 'User';
    var $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'
            )
        ),
        'mot_de_passe' => 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 modèle. Il y a trois manières différentes pour définir les règles de validation : tableaux simples, une règle par champ et plusieurs règles par champ.

> Note des traducteurs francophones : la règle "alphaNumeric" ne fonctionne pas pour notre alphabet, les caractères accentués ne sont pas validés. Pour cela, nous devrons donc passer par une expression régulière.

4.1.1 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 :

```
var $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'utilisateur fourni une adresse email correcte, vous pouvez utiliser cette règle :

```
var $validate = array('utilisateur_email' => 'email');
```

4.1.2 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 :

```
var $validate = array(
    'champ1' => array(
        'rule' => 'nomRegle', // ou bien : array('nomRegle', 'parametre1', 'parametre2'
...)
        '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.

4.1.2.1 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 modèle, 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 :

```
var $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 :

```
var $validate = array(
    'password' => array(
        'rule' => array('minLength', 8)
    )
);
```

Souvenez-vous, la clé 'rule' est obligatoire pour les définitions de règles sous forme de tableau.

4.1.2.2 required

Cette clé doit être définie par une valeur booléenne. Si 'required' est 'true' alors le champ doit être présent dans le tableau de données. Par exemple, si la règle de validation a été définie comme suit :

```
var $validate = array(
    'login' => array(
        'rule' => 'alphaNumeric',
        'required' => true
    )
);
```

Les données envoyées à la méthode save() du modèle 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 ('').

4.1.2.3 allowEmpty

Si définie à 'false', la valeur du champ doit être "non vide", ceci étant déterminé par `!empty($valeur) || is_numeric($valeur)`. 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 modèle, 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.

4.1.2.4 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 nul, la règle s'applique à la fois pendant la création et la mise à jour.

4.1.2.5 message

La clé 'message' vous permet de définir un message d'erreur de validation personnalisé pour la règle :

```
var $validate = array(
    'password' => array(
        'rule' => array('minLength', 8),
        'message' => 'Le mot de passe doit comporter au moins 8 caractères'
    )
);
```

4.1.3 Plusieurs règles par champs

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 oeuvre, 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 modèle.

Si vous souhaitiez affecter plusieurs règles de validation à un seul champ, voici basiquement comment il faudrait faire :

```
var $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 :

```
var $validate = array(
    'login' => array(
        'regle1' => array(
            'rule' => 'alphaNumeric',
```

```
            'message' => 'Lettres et chiffres uniquement'
        ),
        'regle2' => array(
            'rule' => array('minLength', '8'),
            'message' => 'Taille minimum de 8 caractères'
        ),
    )
);
```

L'exemple ci-dessus définit deux règles pour le champ 'login': 'alphanumeric' et 'minlength'. Comme vous pouvez le voir, chaque règle est identifiée avec un nom arbitraire.

Par défaut, CakePHP essaye de valider un champ en utilisant toutes les règles de validation déclarées pour lui et renvoi le message d'erreur pour la dernière règle qui n'est pas passée. Mais si la clé `last` est définie à `true` pour une règle et qu'elle ne passe pas, le message d'erreur pour cette règle est retourné et les règles suivantes ne sont pas validées. Donc si vous préférez voir le message d'erreur de la première règle qui ne passe pas au lieu de la dernière, définissez `'last' => true` pour chaque règle.

Si vous prévoyez d'utiliser des messages d'erreur internationalisé (traduits), vous pouvez définir les messages d'erreur dans vos vues :

```
echo $form->input('login', array(
  'label' => __('Login', true),
  'error' => array(
    'regle1' => __('Caractères alphanumériques seulement', true),
    'regle2' => __('Longueur minimum de 8 caractères', true)
   )
));
```

Le champ est maintenant pleinement internationalisé, et vous pouvez enlever les messages d'erreurs du modèle. Pour plus d'information sur la function `__()`, allez voir la section "Localisation et Internationalisation".

4.1.4 Règles de validation incluses

Les données pour ce champ ne doivent contenir que chiffres et lettres.

```
var $validate = array(
    'login' => array(
        'rule' => 'alphaNumeric',
        'message' => 'Les données pour ce champ ne doivent contenir que lettres et
chiffres.'
    )
);
```

Note des traducteurs francophones : la règle "alphaNumeric" ne fonctionne pas pour notre alphabet, les caractères accentués ne sont pas validés. Pour cela, nous devrons donc passer par une expression régulière.

4.1.4.2 between

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 <.

```
var $validate = array(
    'mot_passe' => array(
        'rule' => array('between', 5, 15),
        'message' => 'Le mot de passe doit avoir une longueur comprise entre 5 et 15
caractères.'
    )
);
```

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.

4.1.4.3 blank

Cette règle est utilisé 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.

```
var $validate = array(
    'id' => array(
        'rule' => 'blank',
        'on' => 'create'
    )
);
```

4.1.4.4 boolean

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'.

```
var $validate = array(
    'maCaseACocher' => array(
        'rule' => array('boolean'),
        'message' => 'Valeur incorrecte pour maCaseACocher'
    )
);
```

4.1.4.5 cc

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.

```
var $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.'
    )
);
```

4.1.4.6 comparison

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 :

```
var $validate = array(
    'age' => array(
        'rule' => array('comparison', '>=', 18),
        'message' => 'Vous devez avoir 18 ans au moins pour vous inscrire.'
);

var $validate = array(
    'age' => array(
        'rule' => array('comparison', 'greater or equal', 18),
        'message' => 'Vous devez avoir 18 ans au moins pour vous inscrire.'
    )
);
```

4.1.4.7 date

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)

Si aucune clé n'est soumise, la clé par défaut 'ymd' sera utilisée.

```
var $validate = array(
    'naissance' => array(
        'rule' => 'date',
        '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 utilisateurs, le mieux ce sera.

4.1.4.8 decimal

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.

```
var $validate = array(
    'prix' => array(
        'rule' => array('decimal', 2)
    )
);
```

4.1.4.9 email

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 soit valide.

```
var $validate = array('email' => array('rule' => 'email'));

var $validate = array(
    'email' => array(
        'rule' => array('email', true),
        'message' => 'Merci de soumettre une adresse email valide.'
    )
);
```

4.1.4.10 equalTo

Cette règle s'assurera que la valeur est égal à la valeur passée et qu'elle est du même type.

```
var $validate = array(
    'nourriture' => array(
        'rule' => array('equalTo', 'gâteau'),
        'message' => 'Cette valeur devrait être la chaîne gâteau'
```

```
    )
);
```

4.1.4.11 extension

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.

```
var $validate = array(
    'image' => array(
        'rule' => array('extension', array('gif', 'jpeg', 'png', 'jpg')),
        'message' => 'Merci de soumettre une image valide.'
    )
);
```

4.1.4.12 file

Cette règle s'assurera qu'une adresse IPv4 valide ait été soumise.

```
var $validate = array(
    'ip_client' => array(
        'rule' => 'ip',
        'message' => 'Merci de soumettre une adresse IP valide.'
    )
);
```

4.1.4.14 isUnique

La donnée pour le champ doit être unique, elle ne peut être utilisée par aucune autre ligne.

```
var $validate = array(
    'login' => array(
        'rule' => 'isUnique',
        'message' => 'Ce nom d\'utilisateur a déjà été
choisi.'
    )
);
```

4.1.4.15 minLength

Cette règle s'assure que la donnée satisfait à la longueur minimale requise.

```
var $validate = array(
    'login' => array(
        'rule' => array('minLength', '8'),
        'message' => 'Les noms d\'utilisateur doivent avoir au moins 8
caractères.'
    )
);
```

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.

4.1.4.16 maxLength

Cette règle s'assure que la donnée respecte la longueur maximale requise.

```
var $validate = array(
    'login' => array(
        'rule' => array('maxLength', '15'),
        'message' => 'Les noms d\'utilisateur ne doivent pas dépasser 15
caractères.'
    )
);
```

La longueur ici est "le nombre d'octets 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.

4.1.4.17 money

Cette règle s'assurera que la valeur est une somme monétaire valide.

Le second paramètre définit où le symbole est situé (gauche/droite).

```
var $validate = array(
    'salaire' => array(
        'rule' => array('money', 'left'),
        'message' => 'Merci de soumettre une somme monétaire valide.'
    )
);
```

4.1.4.18 multiple

Utilisez cette règle pour valider un champ select multiple. Elle accepte les paramètres "in", "max" et "min".

```
var $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'
    )
);
```

4.1.4.19 inList

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 :

```
    var $validate = array(
      'fonction' => array(
          'choixAutorise' => array(
```

```
            'rule' => array('inList', array('Foo', 'Bar')),
            'message' => 'Entrez soit Foo, soit Bar.'
        )
    )
);
```

4.1.4.20 numeric

Vérifie si la donnée passée est un nombre valide.

```
var $validate = array(
    'voitures' => array(
        'rule' => 'numeric',
        'message' => 'Merci de soumettre le nombre de voitures.'
    )
);
```

4.1.4.21 notEmpty

La règle de base pour s'assurer qu'un champ n'est pas vide.

```
var $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".

4.1.4.22 phone

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 additionnels.

```
var $validate = array(
    'telephone' => array(
        'rule' => array('phone', null, 'us')
    )
);
```

4.1.4.23 postal

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.

```
var $validate = array(
    'code_postal' => array(
```

```
        'rule' => array('postal', null, 'us')
    )
);
```

4.1.4.24 range

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.

```
var $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 !!!

4.1.4.25 ssn

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.

```
var $validate = array(
    'ssn' => array(
        'rule' => array('ssn', null, 'us')
    )
);
```

4.1.4.26 url

Cette règle vérifie les formats valides d'URL. Elle supporte les protocoles http(s), ftp(s), file, news et gopher.

```
var $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.

```
var $validate = array(
    'siteweb' => array(
        'rule' => array('url', true)
    )
);
```

4.1.5 Règles personnalisées de validation des données

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.

```
var $validate = array(
    'login' => array(
        'rule' => array('custom', '/^[a-z0-9]{3,}$/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 insensible à la casse.

4.1.5.2 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 :

```
<?php
class Utilisateur extends AppModel {
    var $name = 'Utilisateur';

    var $validate = array(
        'code_promo' => array(
            'rule' => array('limiteUtilisations', 25),
            'message' => 'Ce code promo a dépassé son nombre maximal
d\'utilisation.'
        )
    );

    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' => $data,
'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 modèle (comme dans l'exemple) ou dans un comportement (*behavior*) que votre modèle implémente. Ceci inclus les méthodes mappées.

Les méthodes des modèles/comportements sont vérifiées en premier, avant de chercher pour une méthode dans la class `Validation`. Cela veut dire que vous pouvez écraser les méthodes de validation existantes (telle que `alphaNumeric()`) au niveau de l'application (en ajoutant la méthode dans `AppModel`) ou au niveau du modèle.

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.

```
<?php
class Message extends AppModel {
    var $name = 'Message';
    var $validate = array(
        'pseudo' => array(
            'rule' => 'alphaNumericDashUnderscore',
            'message' => 'Le pseudo ne peut contenir que des lettres, des nombres, des
tirets ou des underscores.'
        )
    );

    function alphaNumericDashUnderscore($check) {
        // le tableau $check est passé en utilisant le nom du champ de formulaire
comme clé
        // nous devons extraire la valeur pour rendre la fonction générique
        $valeur = array_values($data);
        $valeur = $valeur[0];

        return preg_match('|^[0-9a-zA-Z_-]*$|', $valeur);
    }
}
?>
```

4.1.6 Valider des données à partir du contrôleur

Alors que d'habitude, vous utiliser la méthode *save* de votre modèle, il peut vous arriver parfois de vouloir tester et valider vos données sans les sauvegarder. Par exemple, vous souhaitez peut-être afficher des informations complémentaires à l'utilisateur avant de sauvegarder définitivement le tout dans votre base de données. Dans ce cas, uniquement tester les données requiert une technique un peu différente de la sauvegarde habituelle.

Tout d'abord, envoyez les données à votre modèle :

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

Ensuite, pour vérifier que les données sont valides, utilisez la méthode de validation de votre modèle, qui retournera *true* (vrai) si les données sont valides, et *false* (faux) sinon :

```
if ($this->NomDuModele->validates()) {
    // Si les données sont valides
} else {
    // Si elles ne le sont pas
}
```

La méthode de validation appelle la méthode *invalidFields* qui remplit la propriété *validationErrors* du modèle. Elle retourne également ces erreurs comme résultat de son appel :

```
$erreurs = $this->NomDuModele->invalidFields(); // renvoie le tableau d'erreurs
validationErrors
```

Il est important de noter que les données doivent être envoyées au modèle avant de pouvoir être validées. C'est différent de la méthode de sauvegarde (*save*) qui vous permet de les envoyer en tant que paramètre. De même, gardez en mémoire qu'il n'est pas nécessaire d'appeler la méthode de validation avant de sauvegarder vos données, car la méthode *save* validera automatiquement ces données avant de les sauvegarder de manière définitive.

Pour valider vos données dans plusieurs modèles, l'approche suivante peut être utilisée :

```
if ($this->NomDuModele->saveAll($this->data, array('validate' => 'only'))) {
  // Les données sont valides
} else {
  // Les données ne sont pas valides
}
```

Si vous avez validé les données avant sauvegarde, vous pouvez désactiver la validation pour éviter une seconde vérification.

```
if ($this->NomDuModele->saveAll($this->data, array('validate' => false))) {
    // sauvegarde sans validation
}
```

4.2 Sécurisation des données

La classe Sanitize de CakePHP peut être utilisée pour éliminer les données malicieuses soumises par l'utilisateur, ainsi que d'autres informations non souhaitées. Sanitize est une librairie du coeur de CakePHP, elle peut donc être utilisée partout dans votre code, mais la meilleure solution est sans doute de l'utiliser dans les contrôleurs ou les modèles.

Tout ce que vous avez à faire est d'inclure la librairie Sanitize :

```
App::import('Sanitize');
```

Une fois que cela est fait, vous pouvez appelez Sanitize de façon statique.

4.2.1 paranoid

`paranoid(string $string, array $allowedChars);`
Cette fonction vide la chaîne $string de tout caractère non alpha-numérique. La fonction peut ignorer certains caractères en les passant dans le tableau des caractères autorisés $allowedChars.

```
$mauvaiseChaine = ";:<script><html><   // >@@#";
echo Sanitize::paranoid($mauvaiseChaine);
// sortie : scripthtml
```

```
echo Sanitize::paranoid($mauvaiseChaine, array(' ', '@'));
// sortie : scripthtml    @@
```

4.2.2 html

`html(string $string, array $options = array())`
Cette méthode prépare les données entrées par l'utilisateur avant leur rendu HTML. C'est particulièrement utile si vous ne voulez pas que les utilisateurs détruisent vos mises en page, insèrent des images ou des scripts dans le contenu de vos pages HTML. Si l'option $remove est définie à vrai (true), le contenu HTML sera détruit plutôt que transformé en entités HTML.

```
$entreeNonSouhaitee = '<font size="99"
color="#FF0000">SALUT</font><script>...</script>';
echo Sanitize::html($entreeNonSouhaitee);
// sortie : &lt;font size=&quot;99&quot;
color=&quot;#FF0000&quot;&gt;SALUT&lt;/font&gt;&lt;script&gt;...&lt;/script&gt;
echo Sanitize::html($entreeNonSouhaitee, array('remove' => true);
// sortie : SALUT...
```

4.2.3 escape

`escape(string $string, string $connection)`
Utilisé pour échapper les déclarations SQL par l'ajout de *slashes*, en tenant compte de la configuration actuelle du système concernant les magic_quotes_gpc. $connection est le nom de la base concernée par l'échappement, telle que nommée dans votre fichier app/config/database.php .

4.2.4 clean

`Sanitize::clean(mixed $data, mixed $options)`

Cette fonction est un nettoyeur multi-usage de force industrielle, destinée à être utilisée sur des tableaux entiers (comme $this->data, par exemple). La fonction prend un tableau (ou une chaîne) et retourne la version nettoyée. Les opérations de nettoyage suivantes sont exécutées sur chaque élément du tableau (de façon récursive) :

- Les espaces bizarres (incluant 0xCA) sont remplacés par des espaces ordinaires.
- Double vérification des caractères spéciaux et suppression des retours chariot pour une sécurité SQL accrue.
- Ajout de *slashes* pour SQL (appelle simplement la fonction sql exposée précédemment).
- Permutation des anti-slashes saisis par l'utilisateur avec des anti-slashes vérifiés.

L'argument $options peut être une chaîne ou un tableau. Lorsqu'une chaîne est fournie, il s'agit du nom de la connexion 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

L'utilisation de clean() avec des options ressemble à quelque chose comme :

```
$this->data = Sanitize::clean($this->data, array('encode' => false));
```

4.3 Gestion des erreurs

Dans l'éventualité d'une erreur irrécupérable dans votre application, il est courant d'arrêter le processus et d'afficher une page d'erreur à l'utilisateur. Pour vous éviter de coder des traitements spécifiques pour cela dans chacun de vos contrôleurs et composants, vous pouvez utiliser la méthode fournie :

```
$this->cakeError(<string errorType>, [array parameters]);
```

Faire appel à cette méthode affichera une page d'erreur à l'utilisateur et arrêtera tout processus ultérieur de votre application.

`parameters` doit être un tableau de chaînes de caractères. Si le tableau contient des objets (objets `Exception` inclus), ils seront transtypés en chaînes.

CakePHP prédéfinit un lot d'erreur-types, mais pour le moment, la plupart ne sont réellement utiles que par le framework lui-même. Celle qui est la plus utile pour le développeur d'applications est la bonne vielle page d'erreur 404. Elle peut être appelée sans paramètres de la façon suivante :

```
$this->cakeError('error404');
```

Alternativement, vous pouvez forcer la page à signaler que l'erreur s'est produite à une URL spécifique en passant le paramètre `url` :

```
$this->cakeError('error404', array('url' => 'une/autre.url'));
```

Cela devient beaucoup plus utile en étendant le gestionnaire d'erreur et en lui soumettant vos propres types d'erreurs. Les gestionnaires d'erreur personnalisés fonctionnent pratiquement comme les actions d'un contrôleur. Vous allez typiquement attribuer `set()` chaque paramètre passé dans la vue, et ensuite afficher un fichier de vue depuis votre répertoire `app/views/errors`.

Créer un fichier `app/app_error.php` avec la définition suivante :

```
<?php
class AppError extends ErrorHandler {
}
?>
```

Les gestionnaires pour les nouveaux types d'erreurs peuvent être implémentés en ajoutant des méthodes à cette classe. Créez simplement une nouvelle méthode avec le nom que vous voulez utiliser comme type d'erreur.

Prenons l'exemple d'une application qui écrit un certain nombre de fichiers sur le disque et qu'il est approprié d'indiquer les erreurs d'écriture à l'utilisateur. Nous ne voulons pas ajouter de code pour cela dans toutes les

parties de notre application, c'est donc un bon exemple pour utiliser un nouveau type d'erreur.

Ajoutez une nouvelle méthode à votre classe `AppError`. Nous prenons un paramètre appelé `fichier` qui sera le lien vers le fichier que nous n'avons pas réussi à écrire.

```
function impossibleEcrireFichier($params) {
  $this->controller->set('fichier', $params['fichier']);
  $this->_outputMessage('impossible_ecrire_fichier');
}
```

Créez la vue dans `app/views/errors/impossible_ecrire_fichier.ctp`

```
<h2>Impossible d'écrire le fichier</h2>
<p>Nous n'avons pas pu écrire le fichier <?php echo $fichier ?> sur le
disque.</p>
```

et lancez l'erreur dans le contrôleur/composant :

```
$this->cakeError('impossibleEcrireFichier', array('fichier'=>'unnomdefichier'));
```

L'implémentation par défaut de `$this->__outputMessage(<view-filename>)` affichera simplement la vue dans `views/errors/<view-filename>.ctp`. Si vous voulez outrepasser ce comportement, vous pouvez redéfinir `__outputMessage($template)` dans votre classe `AppError`.

4.4 Débogage

`debug($var, $showHTML = false, $showFrom = true)`
La fonction debug() est une fonction disponible globalement qui fonctionne de façon similaire à la fonction php print_r(). La fonction debug() vous permet de visualiser le contenu d'une variable de différentes façons. Tout d'abord, si vous souhaitez que les données soient affichées de façon "HTML-friendly", définissez le second paramètre à vrai (true). La fonction retourne également par défaut la ligne et le fichier d'où elle provient.

Le résultat de cette fonction ne sera affiché que si la variable debug (du core) a été définie à une valeur supérieure à 0.

4.4.2 Utiliser la classe de débogage

Pour utiliser le débogueur, il faut tout d'abord que la variable *Configure::read('debug')* aie une valeur différente de 0.

dump($var)

Dump affiche le contenu d'une variable. La fonction affichera toutes les propriétés et méthodes (s'il en existe) de la variable en question.

```
    $toto = array(1,2,3);

    Debugger::dump($toto);

    //Affichera
```

```
    array(
        1,
        2,
        3
    )

    //Objet simple
    $voiture = new Voiture();

    Debugger::dump($voiture);

    //sortie
    Voiture::
    Voiture::couleur= 'rouge'
    Voiture::marque = 'Toyota'
    Voiture::modele = 'Camry'
    Voiture::kilometrage = '15000'
    Voiture::accelerer()
    Voiture::decelerer()
    Voiture::arreter()
```

log($var, $level = 7)

Crée une description détaillée de la pile au moment de l'exécution. La méthode *log()* affichera les données de façon similaire à *Debugger::dump()*, mais dans le fichier *debug.log* plutôt que dans le *buffer* de sortie. Notez que votre dossier *app/tmp* (et son contenu) doivent avoir les permissions en écriture pour que *log()* fonctionne correctement.

trace($options)
Retourne l'état courant de la pile. Chaque ligne du retour inclue la méthode appelée, y compris quel est le fichier et la ligne dont provient l'appel.

```
    //Dans MessagesController::index()
    pr( Debugger::trace() );

    //sortie
    MessagesController::index() - APP/controllers/downloads_controller.php, line 48
    Dispatcher::_invoke() - CORE/cake/dispatcher.php, line 265
    Dispatcher::dispatch() - CORE/cake/dispatcher.php, line 237
    [main] - APP/webroot/index.php, line 84
```

Ci-dessus, vous avez un exemple du contenu engendré par la méthode *Debugger::trace()* dans un contrôleur. Lire la pile de bas en haut vous montre l'ordre dans lequel les fonctions sont appelées. Dans cet exemple, index.php a appelé *Dispatcher::dispatch()*, qui à son tour a appelé *Dispatcher::_invoke()*. La méthode *_invoke()* à ensuite appelé *MessagesController::index()*. Cette information est pratique quand vous travaillez avec des opérations récursives ou des opérations profondes dans la pile, pour identifier quelles fonctions sont en cours d'exécution quand la fonction trace() est appelée.

excerpt($fichier, $ligne, $contexte)
Extrait un bout du fichier contenu dans $fichier (qui est un chemin absolu), et met en évidence la ligne $ligne avec les $contexte lignes l'entourant.

```
    pr( Debugger::excerpt(ROOT.DS.LIBS.'debugger.php', 321, 2) );

    // affichera ceci :
    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, $context = 2) {</span></code>

        [3] => <span class="code-highlight"><code><span style="color:
#000000">        $data = $lines = array();</span></code></span>
        [4] => <code><span style="color: #000000">        $data =
@explode("\n", file_get_contents($file));</span></code>
    )
```

Bien que cette méthode soit souvent utilisée par la classe en interne, elle peut être utile si vous voulez créer vos propres messages d'erreurs ou *logs* pour une situation donnée.

exportVar($var, $recursion = 0)
Convertit une variable de n'importe quel type en une chaîne de caractère pour l'utiliser dans une sortie de débogage. Cette méthode est également utilisée par une grande partie du débogueur pour des conversions internes de variables, et peut être aussi bien utilisée dans votre propre débogueur.

invoke($debugger)
Remplace le débogueur CakePHP avec un nouveau message d'erreur.

4.4.3 Classe de déboguage

La classe de déboguage est nouvelle dans CakePHP 1.2 et offre toujours plus d'options pour obtenir des informations de déboguage. Elle contient plusieurs fonctions qui sont invoquées par CakePHP, et fournit un affichage de la mémoire, et des fonctions de gestion d'erreur.

La classe de déboguage surcharge les messages d'erreurs par défaut de PHP, en les remplaçant par des messages plus pratiques. Ces nouveaux messages sont utilisées par défaut dans CakePHP. Comme pour toutes les fonctions de déboguage, la variable *Configure::debug* doit avoir une valeur supérieure à 0.

Quand une erreur survient, le débogueur affiche l'erreur sur l'écran, et l'enregistre dans le fichier error.log. Le rapport d'erreur est un affichage à la fois de la pile d'instructions et de l'extrait du code près de l'erreur. Cliquez sur le lien "Error" pour afficher la pile, et sur "Code" pour avoir les lignes mises en cause dans l'erreur.

4.5 Mise en cache

Bien que les réglages de la Classe Configure du coeur de CakePHP puissent vraiment vous aider à voir ce qui se passe en arrière plan, 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.

La journalisation (*logging*) 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 mes utilisateurs ont-il vues ? 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 (Modèle, Contrôleur, Composant... n'importe quoi d'autre), vous pouvez journaliser vos données. Vous pouvez aussi utiliser CakeLog::write() directement.

4.6.1 Utiliser la fonction log

La fonction log() prend deux paramètres. Le premier est le message que vous aimeriez écrire dans le fichier de log. Par défaut, ce message d'erreur est écrit dans le fichier de log "error", qui se trouve dans app/tmp/logs/error.log.

```
// Exécuter ceci dans une classe CakePHP :

$this->log("Quelque chose ne marche pas !");

// Entraîne un ajout dans app/tmp/logs/error.log

2007-11-02 10:22:02 Error: Quelque chose ne marche pas !
```

Le second paramètre est utilisé pour définir le type de fichier de log dans lequel vous souhaitez écrire le message. S'il n'est pas précisé, le type par défaut est LOG_ERROR, qui écrit dans le fichier de log "error" mentionné précédemment. Vous pouvez définir ce second paramètre à LOG_DEBUG pour écrire vos messages dans un fichier de log alternatif situé dans app/tmp/logs/debug.log :

```
// Exécuter ceci dans une classe CakePHP :

$this->log('Un message de débug', LOG_DEBUG);

// Entraîne un ajout dans app/tmp/logs/debug.log (plutôt que dans error.log)

2007-11-02 10:22:02 Error: un message de débug.
```

Vous pouvez aussi spécifier un nom différent pour le fichier de log, en définissant le second paramètre avec le nom de ce fichier.

```
// Exécuter ceci dans une classe CakePHP :

$this->log("Un message spécial pour la journalisation de l'activité",
'activite');

// Entraîne un ajout dans app/tmp/logs/activite.log (plutôt que dans
error.log)

2007-11-02 10:22:02 Activite: Un message spécial pour la journalisation de
l'activité
```

Votre dossier app/tmp doit être accessible en écriture par le serveur web pour que la journalisation fonctionne correctement.

4.6.2 Using the default FileLog class

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

While CakeLog can be configured to write to a number of user configured logging adapters, it also comes with a default logging configuration. This configuration is identical to how CakeLog behaved in CakePHP 1.2. The default logging configuration will be used any time there are *no other* logging adapters configured. Once a logging adapter has been configured you will need to also configure FileLog if you want file logging

to continue.

As its name implies FileLog writes log messages to files. The type of log message being written determines the name of the file the message is stored in. If a type is not supplied, LOG_ERROR is used which writes to the error log. The default log location is `app/tmp/logs/$type.log`

```
//Executing this inside a CakePHP class:
 $this->log("Something didn't work!");

//Results in this being appended to app/tmp/logs/error.log
2007-11-02 10:22:02 Error: Something didn't work!
```

You can specify a custom log names, using the second parameter. The default built-in FileLog class will treat this log name as the file you wish to write logs to.

```
//called statically
CakeLog::write('activity', 'A special message for activity logging');

//Results in this being appended to app/tmp/logs/activity.log (rather than error.log)
2007-11-02 10:22:02 Activity: A special message for activity logging
```

The configured directory must be writable by the web server user in order for logging to work correctly.

You can configure additional/alternate FileLog locations using `CakeLog::config()`. FileLog accepts a `path` which allows for custom paths to be used.

```
CakeLog::config('custom_path', array(
    'engine' => 'FileLog',
    'path' => '/path/to/custom/place/'
));
```

4.6.3 Creating and configuring log streams

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Log stream handlers can be part of your application, or part of plugins. If for example you had a database logger called `DataBaseLogger` as part of your application it would be placed in `app/libs/log/data_base_logger.php`; as part of a plugin it would be placed in `app/plugins/my_plugin/libs/log/data_base_logger.php`. When configured, `CakeLog` will attempt to load. Configuring log streams is done by calling `CakeLog::config()`. Configuring our DataBaseLogger would look like

```
//for app/libs
CakeLog::config('otherFile', array(
    'engine' => 'DataBaseLogger',
    'model' => 'LogEntry',
    ...
));
```

```
//for plugin called LoggingPack
CakeLog::config('otherFile', array(
    'engine' => 'LoggingPack.DataBaseLogger',
    'model' => 'LogEntry',
    ...
));
```

When configuring a log stream the `engine` parameter is used to locate and load the log handler. All of the other configuration properties are passed to the log stream's constructor as an array.

```
class DataBaseLogger {
    function __construct($options = array()) {
        //...
    }
}
```

CakePHP has no requirements for Log streams other than that they must implement a `write` method. This write method must take two parameters `$type, $message` in that order. `$type` is the string type of the logged message, core values are `error`, `warning`, `info` and `debug`. In addition you can define your own types by using them when you call `CakeLog::write`.

It should be noted that you will encounter errors when trying to configure application level loggers from `app/config/core.php`. This is because paths are not yet bootstrapped. Configuring of loggers should be done in `app/config/bootstrap.php` to ensure classes are properly loaded.

4.6.4 Interacting with log streams

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

You can introspect the configured streams with `CakeLog::configured()`. The return of `configured()` is an array of all the currently configured streams. You can remove streams using `CakeLog::drop($key)`. Once a log stream has been dropped it will no longer receive messages.

4.6.5 Error logging

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Errors are now logged when `Configure::write('debug', 0);`. You can use `Configure::write('log', $val)`, to control which errors are logged when debug is off. By default all errors are logged.

```
Configure::write('log', E_WARNING);
```

Would log only warning and fatal errors. Setting `Configure::write('log', false);` will disable error logging when debug = 0.

4.7 Tester

A partir de la version 1.2, CakePHP inclus le support d'un framework de test global. Ce framework est une extension du framework SimpleTest pour PHP. Cette section exposera comment préparer, construire et

exécuter vos tests.

4.7.1 Préparation aux tests

Le framework de tests fourni avec CakePHP 1.2 est basé sur le framework de tests SimpleTest. SimpleTest n'est pas livré avec l'installation par défaut de CakePHP, donc nous avons d'abord besoin de le télécharger. Vous pouvez le trouver ici : http://simpletest.sourceforge.net/.

Récupérez la dernière version et décompressez le code dans votre dossier **`cake/vendors`** ou votre dossier **`app/vendors`**, comme vous préférez. Vous devriez maintenant avoir un répertoire **`vendors/simpletest`** avec tous les fichiers et dossiers de SimpleTest à l'intérieur. Pensez à mettre le niveau de DEBUG à 1 au moins dans votre fichier **`app/config/core.php`** avant de lancer vos tests !

Si vous n'avez pas défini de connexion à une base de données de test dans votre fichier **`app/config/database.php`**, les tables de test seront créées avec le préfixe `test_suite_`. Vous pouvez créer une connexion à la base de données `$test` comme celle présentée ci-dessous, pour contenir toutes les tables de test :

```
    var $test = array(
        'driver' => 'mysql',
        'persistent' => false,
        'host' => 'serveur',
        'login' => 'identifiant',
        'password' => 'mot_passe',
        'database' => 'nomBase'
    );
```

Si la base de données test est disponible et que CakePHP peut s'y connecter, toutes les tables y seront créées.

4.7.1.2 Lancer les cas de tests du coeur

Les différents packages de CakePHP 1.2 ne sont pas livrés avec les cas de test du coeur. Pour obtenir ces tests, vous devez les télécharger depuis le dépôt. Toutes les versions de CakePHP sont actuellement situées sur le site http://code.cakephp.org/. Vous aurez besoin de vous créer un compte utilisateur avec une clé personnalisée et utiliser Git pour accéder au dépôt.

Pour ajouter les tests du coeur à votre application existante, décompressez le package *nightly* téléchargé dans un répertoire temporaire. Localisez le répertoire `/cake/tests` sur le dépôt et copiez-le (récursivement) dans votre dossier `/cake/tests`.

Ces tests peuvent alors être atteints en naviguant à l'adresse http://votre.domaine.cake/test.php - qui dépend de vos paramétrages spécifiques. Essayez d'exécuter l'un des groupes de test du coeur en cliquant sur le lien correspondant. L'exécution d'un groupe de tests peut prendre un certain temps, mais vous devriez finir par voir quelque chose comme : "2/2 test cases complete: 49 passes, 0 fails and 0 exceptions".

Félicitations, vous êtes maintenant prêt à commencer l'écriture de tests !

Si vous lancez tous les tests du coeur ensemble ou les groupes de tests, la plupart échoueront. Ceci est connu des développeurs CakePHP et normal, donc pas de panique. Essayez plutôt de lancer chacun des cas de test du coeur individuellement.

4.7.2 Vue d'ensemble du Test - Test unitaire vs Test Web

Le framework de test CakePHP propose 2 façons de tester. L'une est le Test Unitaire, où vous testez de petites parties de votre code, comme une méthode dans un composant ou une action dans un contrôleur. L'autre type de test supporté est le Test Web, où vous automatisez le travail de test de votre application, à travers la navigation entre pages, le remplissage de formulaires, le clic sur des liens, etc.

4.7.3 Préparation des données de test

Quand on test du code qui dépend des modèles et des données, on peut utiliser les **fixtures** (garnitures) comme moyen de générer temporairement des tables, remplies avec des échantillons de données qui peuvent être utilisés par le test. Le bénéfice apporté par l'utilisation des fixtures, c'est que votre test n'a aucune chance de corrompre les données de l'application en production. En plus, vous pouvez commencer à tester votre code, avant de développer le contenu réel d'une application.

CakePHP tente d'utiliser la connexion nommée `$test` dans votre fichier de configuration **`app/config/database.php`**. Si cette connexion n'est pas utilisable, il utilisera la configuration de base de données `$default` et créera les tables de test dans la base de données définie par cette configuration. Dans les deux cas, il ajoutera "test_suite_" à vos propres préfixes de table (s'il y en a) pour éviter des conflits avec vos tables existantes.

CakePHP réalise ce qui suit durant l'utilisation d'un cas de test basé sur une fixture :

1. Créer les tables nécessaires à chaque fixture
2. Remplir les tables avec des données, si les données sont fournies dans la fixture
3. Lancer les méthodes de test
4. Vider les tables de fixtures
5. Supprimer les tables de fixture de la base de données

4.7.3.2 Créer des fixtures

En créant une *fixture*, vous définirez principalement deux choses : comment la table est composée (quels champs font partie de la table) et quels enregistrements seront initialement remplis dans la table de test. Créons notre première *fixture*, qui sera utilisée pour tester notre propre modèle Article. Créez un fichier nommé **article_fixture.php** dans votre répertoire **app/tests/fixtures**, avec le contenu suivant :

```
<?php
 class ArticleFixture extends CakeTestFixture {
      var $name = 'Article';

      var $fields = array(
          'id' => array('type' => 'integer', 'key' => 'primary'),
          'titre' => array('type' => 'string', 'length' => 255, 'null' =>
false),
          'contenu' => 'text',
          'publiable' => array('type' => 'integer', 'default' => '0', 'null'
=> false),
          'created' => 'datetime',
          'updated' => 'datetime'
      );
      var $records = array(
          array ('id' => 1, 'titre' => 'Premier Article', 'contenu' => 'Corps du
premier Article', 'publiable' => '1', 'created' => '2007-03-18 10:39:23', 'updated'
=> '2007-03-18 10:41:31'),
          array ('id' => 2, 'titre' => 'Second Article', 'contenu' => 'Corps du
second Article', 'publiable' => '1', 'created' => '2007-03-18 10:41:23', 'updated'
=> '2007-03-18 10:43:31'),
```

```
        array ('id' => 3, 'titre' => 'Troisième Article', 'contenu' =>
'Corps du troisième Article', 'publiable' => '1', 'created' => '2007-03-18
10:43:23', 'updated' => '2007-03-18 10:45:31'),
    );
 }
 ?>
```

La variable `$name` est extrêmement importante. Si vous l'omettez, cake utilisera un mauvais nom de table lorsqu'il génèrera votre base de donnée de test, et vous aurez alors d'étranges erreurs qui seront difficiles à débugger. Si vous utilisez PHP 5.2, vous pouvez utiliser des classes de modèle sans `$name`, mais vous devez penser à l'inclure dans vos fichiers *fixture*.

Nous utilisons $fields pour spécifier quels champs feront partie de cette table et comment seront-ils définis. Le format utilisé pour définir ces champs est le même que celui utilisé dans la fonction **generateColumnSchema()** définie dans les classes de moteur de base de données de Cake (par exemple, dans le fichier dbo_mysql.php.) Voyons les attributs disponibles qu'un champ peut prendre et leur signification :

type
: Type interne de donnée CakePHP. Supportés actuellement : string (correspond à VARCHAR), text (correspond à TEXT), integer (correspond à INT), float (correspond à FLOAT), datetime (correspond à DATETIME), timestamp (correspond à TIMESTAMP), time (correspond à TIME), date (correspond à DATE) et binary (correspond à BLOB)

key
: définir à *primary* pour rendre le champ AUTO_INCREMENT et PRIMARY KEY de la table.

length
: définir à la longueur spécifique que le champ devrait prendre.

null
: définir soit à *true* (pour autoriser les champs NULLs) ou à *false* (pour les interdire)

default
: valeur par défaut que le champ doit prendre.

Nous pouvons enfin définir un ensemble d'enregistrements qui seront remplis après que la table de test soit créée. Le format est moyennement explicite et nécessite un peu plus d'explication. 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 nécessite d'avoir une valeur NULL, spécifiez simplement la valeur de cette clé à NULL.

4.7.3.3 Importer les informations de la table et des enregistrements

Votre application peut avoir déjà fait travailler les modèles avec de vraies données associées et vous pourriez décider de tester votre modèle avec ces données. Il pourrait être alors redondant, d'avoir à définir la structure de la table et/ou les enregistrements dans vos fixtures. Heureusement, il y a une manière pour vous de définir cette structure de table et/ou les enregistrements pour une fixture particulière provenant d'un modèle existant ou d'une table existante.

Commençons par un exemple. Considérant que vous avez un modèle nommé Article disponible dans votre application (qui correspond à la table nommée articles), changez la fixture donnée en exemple dans la section précédente (**app/tests/fixtures/article_fixture.php**) de cette façon :

```
<?php
  class ArticleFixture extends CakeTestFixture {
        var $name = 'Article';
        var $import = 'Article';
  }
```

```
    ?>
```

Cette déclaration indique à la suite de tests d'importer la définition de votre table liée au modèle nommé Article. Vous pouvez utiliser tout modèle disponible dans votre application. La déclaration ci-dessus n'importe pas d'enregistrements, vous pouvez donc le faire en la modifiant ainsi :

```
class ArticleFixture extends CakeTestFixture {
    var $name = 'Article';
    var $import = array('model' => 'Article', 'records' => true);
}
?>
```

Si, au contraire, vous avez une table créée mais pas de modèle disponible pour elle, vous pouvez préciser que votre import s'effectuera en lisant cette information de table à la place. Par exemple :

```
<?php
  class ArticleFixture extends CakeTestFixture {
         var $name = 'Article';
         var $import = array('table' => 'articles');
  }
?>
```

Importera la définition de table appelée 'articles' en utilisant votre connexion CakePHP à la base de données nommée 'default'. Si vous voulez changer la connexion à utiliser, faites simplement :

```
<?php
  class ArticleFixture extends CakeTestFixture {
  var $name = 'Article';
  var $import = array('table' => 'articles', 'connection' => 'autre');
  }
  ?>
```

Puisqu'elle utilise votre connexion CakePHP à la base de données, s'il y a un quelconque préfixe de table déclaré, il sera automatiquement utlisé quand vous récupérerez les informations de la table. Les deux fragments ci-dessus n'importent pas les enregistrements de la table. Pour forcer la fixture à importer aussi ses enregistrements, changez-la en :

```
<?php
  class ArticleFixture extends CakeTestFixture {
         var $name = 'Article';
         var $import = array('table' => 'articles', 'records' => true);
  }
?>
```

Vous pouvez naturellement importer votre structure de table depuis une table/un modèle existant, mais avoir défini vos enregistrements directement dans la fixture, comme ce fut montré dans la section précédente. Par exemple :

```
<?php
```

```
   class ArticleFixture extends CakeTestFixture {
          var $name = 'Article';
          var $import = 'Article';

          var $records = array(
              array ('id' => 1, 'titre' => 'Premier Article', 'contenu' => 'Corps
du premier Article', 'publiable' => '1', 'created' => '2007-03-18 10:39:23',
'updated' => '2007-03-18 10:41:31'),
              array ('id' => 2, 'titre' => 'Second Article', 'contenu' => 'Corps
du second Article', 'publiable' => '1', 'created' => '2007-03-18 10:41:23',
'updated' => '2007-03-18 10:43:31'),
              array ('id' => 3, 'titre' => 'Troisième Article', 'contenu'
=> 'Corps du troisième Article', 'publiable' => '1', 'created' =>
'2007-03-18 10:43:23', 'updated' => '2007-03-18 10:45:31'),
          );
   }
 ?>
```

4.7.4 Créer des tests

D'abord, revoyons un certain nombre de règles ou directives concernant les tests :

1. Les fichiers PHP contenant des tests devraient être dans votre dossier : **app/tests/cases/[un_dossier]**.
2. Les noms de ces fichiers devraient se terminer en **.test.php** plutôt que simplement .php.
3. Les classes contenant les tests devraient étendre **CakeTestCase** ou **CakeWebTestCase**.
4. Le nom de toute méthode contenant un test (par ex. contenant une assertion) devrait commencer par **test**, comme dans **testPublished()**.

Quand vous avez créé un cas de test, vous pouvez l'exécuter en naviguant à l'adresse **http://votre.domaine.cake/dossier_cake/test.php** (ceci dépend de vos réglages spécifiques) et cliquer les cas de test de l'App, puis cliquer le lien vers votre fichier spéficique.

4.7.4.1 CakeTestCase Méthodes Callback

Si vous voulez effectuer quelques opérations juste avant ou après un CakeTestCase individuel, et/ou avant ou après le CakeTestCase entiern les méthodes "callbacks" suivants sont disponibles :

start()
Première méthode appelée dans un *test case*.

end()
Dernière méthode appelée dans un *test case*.

startCase()
Appelée avant le démarrage d'un *test case*.

endCase()
Appelée après la fin d'un *test case*.

before($methode)
Annonce le démarrage de la méthode "$methode" du *test case* en cours.

after($methode)
Annonce la fin de la méthode "$methode" du *test case* en cours.

startTest($methode)
Appelée juste avant le démarrage de la méthode "$methode" du *test case* en cours.

endTest($methode)
Appelée juste après la fin de la méthode "$methode" du *test case* en cours.

4.7.5 Tester les modèles

Disons que nous avons déjà notre modèle Article défini dans app/models/article.php, lequel ressemble à ceci :

```
<?php
  class Article extends AppModel {
          var $name = 'Article';

          function publiable($champs = null) {
              $conditions = array(
                  $this->name . '.publiable' => 1
              );

              return $this->find('all',array(
                  'conditions'=>$conditions,
                  'fields'=>$champs
                ));
          }

  }
?>
```

Nous voulons maintenant définir un test qui utilisera cette définition de modèle, mais à travers des *fixtures*, pour tester quelques fonctionnalités du modèle. La suite de test CakePHP charge un tout petit ensemble de fichiers (pour garder les tests isolés), ainsi nous commençons par charger notre modèle parent (dans ce cas le modèle Article que nous avons déjà défini), puis par informer la suite de test que nous voulons tester ce modèle, en précisant quelle configuration de bases de données elle devrait utiliser. La suite de test CakePHP fournit une configuration de BDD nommée **test_suite** qui est utilisée pour tous les modèles reliés aux *fixtures*. Définir $useDbConfig pour cette configuration indiquera à CakePHP que ce modèle utilise la connexion à la base de données de la suite de test.

Les modèles CakePHP utiliseront seulement la configuration de BDD test_suite s'ils sont reliés à des *fixtures* dans votre cas de test !

Puisque nous voulons également réutiliser tout le code existant de notre modèle, nous allons créer un test de modèle qui étendra Article et définir $useDbConfig et $name judicieusement. Créons maintenant un fichier nommé **article.test.php** dans votre répertoire **app/tests/cases/models**, avec le contenu suivant :

```
<?php
  App::import('Model','Article');


  class ArticleTestCase extends CakeTestCase {
          var $fixtures = array( 'app.article' );
  }
?>
```

Nous avons créé le cas de test ArticleTestCase. Dans la variable **$fixtures** nous définissons un ensemble de *fixtures* que nous utiliserons.

Si votre modèle est associé avec d'autres modèles, vous devrez inclure TOUTES les *fixtures* pour chaque modèle associé, même si vous ne les utilisez pas. Par exemple : A hasMany B hasMany C hasMany D. Dans ATestCase vous devrez inclure les *fixtures* pour a, b, c et d.

4.7.5.2 Créer une méthode de test

Ajoutons maintenant une méthode pour tester la fonction publiable() du modèle Article. Editer le fichier **app/tests/cases/models/article.test.php** afin qu'il ressemble désormais à ceci :

```
  <?php
    App::import('Model','Article');

    class ArticleTestCase extends CakeTestCase {
        var $fixtures = array( 'app.article' );

        function testPubliable() {
            $this->Article =& ClassRegistry::init('Article');

            $resultat = $this->Article->publiable(array('id', 'titre'));
            $attendus = array(
                array('Article' => array( 'id' => 1, 'titre' => 'Premier Article'
)),
                array('Article' => array( 'id' => 2, 'titre' => 'Second Article'
)),
                array('Article' => array( 'id' => 3, 'titre' => 'Troisième
Article' )),
            );

            $this->assertEqual($resultat, $attendus);
        }
    }
    ?>
```

Vous pouvez voir que nous avons ajouté une méthode appelée **testPubliable()**. Nous commençons par créer une instance de notre *fixture* basée sur le modèle **Article**, puis nous lançons notre méthode **publiable()**. Dans **$attendus**, nous définissons ce que nous estimons comme devant être le résultat correct (ce que nous savons, puisque nous avons défini quels enregistrements sont initialement remplis dans la table article). Nous vérifions que le résultat est égal à notre prévision, en utilisant la méthode **assertEqual**. Voyez la section Créer des Tests pour plus d'information sur la manière de lancer le test.

4.7.6 Tester les contrôleurs

Imaginons que vous ayez un contrôleur typique articles, avec son modèle correspondant et qui ressemble à ceci :

```
<?php
class ArticlesController extends AppController {
   var $name = 'Articles';
   var $helpers = array('Ajax', 'Form', 'Html');

   function index($short = null) {
     if (!empty($this->data)) {
       $this->Article->save($this->data);
     }
     if (!empty($short)) {
       $resultat = $this->Article->findAll(null, array('id', 'titre'));
     } else {
```

```
        $resultat = $this->Article->findAll();
      }

      if (isset($this->params['requested'])) {
        return $resultat;
      }

      $this->set('titre', 'Articles');
      $this->set('articles', $resultat);
    }
}
?>
```

Créez un fichier nommé articles_controller.test.php dans votre répertoire app/tests/cases/controllers et mettez-y ce qui suit :

```
<?php
class ArticlesControllerTest extends CakeTestCase {
    function startCase() {
      echo '<h1>Démarrage du Cas de Test</h1>';
    }
    function endCase() {
      echo '<h1>Fin du Cas de Test</h1>';
    }
    function startTest($methode) {
      echo '<h3>Début de la méthode ' . $methode . '</h3>';
    }
    function endTest($methode) {
      echo '<hr />';
    }
    function testIndex() {
      $resultat = $this->testAction('/articles/index');
      debug($resultat);
    }
    function testIndexShort() {
      $resultat = $this->testAction('/articles/index/short');
      debug($resultat);
    }
    function testIndexShortGetRenderedHtml() {
      $resultat = $this->testAction('/articles/index/short', array('return' =>
'render'));
      debug(htmlentities($resultat));
    }
    function testIndexShortGetViewVars() {
      $resultat = $this->testAction('/articles/index/short', array('return' =>
'vars'));
      debug($resultat);
    }
    function testIndexFixturized() {
      $resultat = $this->testAction('/articles/index/short', array('fixturize' =>
true));
      debug($resultat);
    }
    function testIndexPostFixturized() {
      $data = array('Article' => array('user_id' => 1, 'publiable' => 1,
'slug'=>'nouvel-article', 'titre' => 'Nouvel Article', 'contenu' => 'Nouveau
Contenu'));
      $resultat = $this->testAction('/articles/index', array('fixturize' => true,
'data' => $data, 'method' => 'post'));
      debug($resultat);
    }
}
```

```
?>
```

4.7.6.2 La méthode testAction

La chose nouvelle ici, c'est la méthode **testAction**. Le premier argument de cette méthode est l'url Cake de l'action du contrôleur à tester, comme dans '/articles/index/short'.

Le second argument est un tableau de paramètres, composé de :

return
: Définir à la valeur que vous voulez retourner.
Les valeurs possibles sont :
◊ 'vars' - Vous obtenez les variables de la vue disponible après l'éxécution de l'action
◊ 'view' - Vous obtenez la vue générée, sans le layout
◊ 'contents' - Vous obtenez la vue HTML complète, incluant le layout
◊ 'result' - Vous obtenez la valeur retournée quand l'action utilise $this->params['requested'].
Par défault c'est 'result'.

fixturize
: Définir à vrai si vous voulez que vos modèles soient auto-fixturisés (ainsi les tables de votre application seront copiées, avec leurs enregistrements, pour les tester sans affecter votre application réelle en cas de modification des données). Si vous définissez 'fixturize' comme un tableau de modèles, alors seuls ces modèles seront auto-fixturisés, tandis que les autres conserveront leurs vraies tables. Si vous souhaitez utiliser vos fichiers de fixture avec testAction(), n'utilisez pas fixturize, mais plutôt les fixtures comme vous l'auriez fait normalement.

method
: définir à 'post' ou 'get' si vous voulez passez des données au contrôleur

data
: les données à passer. A définir comme un tableau associatif composé de champs => valeur. Jetez un oeil à `function testIndexPostFixturized()` dans le cas de test plus haut, pour voir comment nous émulons le postage des données d'un formulaire, lors de la soumission d'un nouvel article.

4.7.6.3 Pièges

Si vous utilisez testAction pour tester une méthode de l'un de vos contrôleurs qui fait une redirection, votre test se terminera immédiatement, sans retourner aucun résultat.
Voyez https://trac.cakephp.org/ticket/4154 pour une possible correction.

4.7.7 Tester les Assistants

Puisqu'un pourcentage respectable de la logique réside dans les classes Assistant (*Helper*), il est important de s'assurer que ces classes sont couvertes par les cas de test.

Le test d'Assistant est un brin similaire à l'approche utilisée pour les Composants. Supposez que nous ayons un assistant appelé InterpreteurDeMonnaieHelper situé dans `app/views/helpers/interpreteur_de_monnaie_helper.php` accompagné de son fichier de cas de test situé dans `app/tests/cases/helpers/interpreteur_de_monnaie_helper.test.php`

4.7.7.1 Créer un test d'Assistant, 1ère partie

Pour commencer, nous définirons les responsabilités de notre assistant DevisesFormateurHelper. En gros, il aura deux méthodes, juste pour les besoins de la démonstration :

```
function dollar($montant)
```

Cette fonction recevra le montant à formater. Celui-ci prendra 2 décimales, avec les espaces manquants remplis par des zéros et le préfixe 'USD' ajouté.

`function euro($montant)`
Cette fonction fera la même chose que dollar() mais préfixera le montant retourné avec 'EUR'. Juste pour rendre le tout un peu plus complexe, nous allons aussi entourer le résultat par des balises span :

```
<span class="euro"></span>
```

Créons d'abord les tests :

```
<?php

//Importe l'assistant à tester.
//Si l'assistant testé doit utiliser d'autres assistants, comme Html,
//ils devront être importés dans cette ligne et instanciés dans
startTest().
App::import('Helper', 'DevisesFormateur');

class DevisesFormateurTest extends CakeTestCase {
    private $devisesFormateur = null;

    //Ici nous instancions notre assistant et tous les autres dont nous avons besoin.
    public function startTest() {
        $this->devisesFormateur = new DevisesFormateurHelper();
    }

    //test de la fonction dollar().
    public function testDollar() {
        $this->assertEqual('USD 5,30', $this->devisesFormateur->dollar(5,30));
        //On devrait toujours avoir deux chiffres après la virgule.
        $this->assertEqual('USD 1,00', $this->devisesFormateur->dollar(1));
        $this->assertEqual('USD 2,05', $this->devisesFormateur->dollar(2,05));
        //Test du séparateur de milliers
        $this->assertEqual('USD 12 000,70', $this->devisesFormateur->dollar(12
000,70));
    }
}
```

Ici, nous appelons `dollar()` avec différents paramètres et nous demandons à la suite de test de vérifier si les valeurs retournées sont égales à ce qui est attendu.

Exécuter le test maintenant provoquera des erreurs (parce que DevisesFormateurHelper n'existe même pas) montrant que nous avons 3 échecs.

Une fois que nous savons ce que notre méthode devrait faire, nous pouvons écrire la méthode elle-même :

```
<?php
class DevisesFormateurHelper extends AppHelper {
    public function dollar($montant) {
        return 'USD ' . number_format($montant, 2, ',', ' ');
    }
}
```

Ici nous définissons le nombre de décimales à 2, le séparateur de décimale à la virgule, le séparateur des

milliers à l'espace et le préfixe du nombre formaté à la chaîne 'USD'.

Enregistrez çà dans **app/views/helpers/devises_formateur.php** et exécutez le test. Vous devriez voir une barre verte et un message indiquant 4 passes.

4.7.8 Tester les composants

Considérons que nous voulions tester un composant appelé TransporteurComponent, qui utilise un modèle appelé Transporteur pour fournir des fonctionnalités aux autres contrôleurs. Nous utiliserons quatre fichiers :

- Un composant appelé Transporteur situé dans **app/controllers/components/transporteur.php**
- Un modèle appelé Transporteur situé dans **app/models/transporteur.php**
- Une fixture appelée TransporteurTestFixture située dans **app/tests/fixtures/transporteur_fixture.php**
- Le code du test situé dans **app/tests/cases/transporteur.test.php**

4.7.8.1 Initialiser le composant

Puisque CakePHP déconseille d'importer les modèles directement dans les composants, nous avons besoin d'un contrôleur pour accéder aux données dans le modèle.

Si la fonction startup() du composant ressemble à ceci :

```
public function startup(&$controller){
          $this->Transporteur = $controller->Transporteur;
 }
```

alors nous pouvons simplement définir une fausse classe vraiment toute simple :

```
class FauxTransporteurController {}
```

et lui assigner des valeurs comme çà :

```
$this->TransporteurComponentTest = new TransporteurComponent();
$controller = new FauxTransporteurController();
$controller->Transporteur = new TransporteurTest();
$this->TransporteurComponentTest->startup(&$controller);
```

4.7.8.2 Créer une méthode de test

Créez simplement une classe qui étende CakeTestCase et commencez à écrire des tests !

```
class TransporteurTestCase extends CakeTestCase {
    var $fixtures = array('transporteur');
    function testGetTransporteur() {
          $this->TransporteurComponentTest = new TransporteurComponent();
          $controller = new FauxTransporteurController();
          $controller->Transporteur = new TransporteurTest();
          $this->TransporteurComponentTest->startup(&$controller);

          $resultat = $this->TransporteurComponentTest->getTransporteur("12345",
"Suéde", "54321", "Suède");
```

```
            $this->assertEqual($resultat, 1, "SP est meilleur pour 1xxxx-5xxxx");

            $resultat = $this->TransporteurComponentTest->getTransporteur("41234",
"Suéde", "44321", "Suède");
            $this->assertEqual($resultat, 2, "WSTS est meilleur pour 41xxx-44xxx");

            $resultat = $this->TransporteurComponentTest->getTransporteur("41001",
"Suéde", "41870", "Suède");
            $this->assertEqual($resultat, 3, "GL est meilleur pour 410xx-419xx");

            $resultat = $this->TransporteurComponentTest->getTransporteur("12345",
"Suéde", "54321", "Norvège");
            $this->assertEqual($resultat, 0, "Aucun ne peut desservir la
Norvège");
    }
}
```

4.7.9 Test Web - Tester les vues

La plupart du temps, si ce n'est toujours, les projets CakePHP sont des applications web. Tandis que les tests unitaires sont un excellent moyen de tester de petites parties d'une fonctionnalité, vous pourriez aussi vouloir tester la fonctionnalité à plus large échelle. La classe **CakeWebTest**Case offre une bonne méthode pour réaliser ce test du point de vue de l'utilisateur.

4.7.9.1 A propos de CakeWebTestCase

CakeWebTestCase est une extension directe du SimpleTest WebTestCase, sans aucune autre fonctionnalité. Toutes les fonctionnalités trouvées dans SimpleTest, documentation pour le test Web est également disponible ici. Cela veut dire aussi, qu'aucune autre fonctionnalité que celles de SimpleTest n'est disponible. Cela veut dire que vous ne pouvez pas utiliser les fixtures et que **tous les cas de test web réclamant des mises à jour/sauvegardes dans la base de données, changeront de manière permanente les valeurs de votre base de données**. Les résultats de test sont souvent basés sur les valeurs que contient la base de données, donc s'assurer que la base contient des valeurs que vous attendez fait partie de la procédure de test.

4.7.9.2 Créer un test

Pour être en conformité avec les autres conventions de test, vous devriez créer vos tests de vue dans tests/cases/views. Vous pouvez, bien sÃ»r, mettre ces tests n'importe où, mais suivre les conventions chaque fois que c'est possible est toujours une bonne idée. Créons le fichier tests/cases/views/web_complet.test.php

D'abord, lorsque vous voulez écrire des tests web, vous devez penser à étendre **CakeWebTestCase** plutôt que CakeTestCase :

```
class WebCompletTestCase extends CakeWebTestCase
```

Si vous avez besoin de faire quelques préparatifs avant que vous ne commenciez le test, créez un constructeur :

```
function WebCompletTestCase(){
  //Faire des trucs ici
}
```

En écrivant les vrais cas de test, la première chose que vous devez faire est d'obtenir quelque chose à regarder. Cela peut se faire en réalisant une requête **get** ou **post**, en utilisant respectivement **get()** ou **post()**. Ces deux méthodes prennent une url absolue comme premier paramètre. Ceci peut être récupéré dynamiquement, si nous supposons que le script de test est situé sous http://votre.domaine/cake/folder/webroot/test.php, en tapant :

```
$this->baseurl = current(split("webroot", $_SERVER['PHP_SELF']));
```

Vous pouvez alors faire des gets et des posts en utilisant les urls Cake, comme ceci :

```
$this->get($this->baseurl."/produits/index/");
$this->post($this->baseurl."/client/login", $data);
```

Le second paramètre de la méthode post , **$data**, est un tableau associatif contenant les données postées au format Cake :

```
$data = array(
  "data[Client][mail]" => "utilisateur@utilisateur.com",
  "data[Client][mot_passe]" => "passeutilisateur");
```

Quand vous avez requêté la page, vous pouvez faire toutes sortes d'actions dessus, en utilisant les méthodes standard de test web de SimpleTest.

4.7.9.3 Parcourir une page

Les tests pour plugins sont créés dans leur propre répertoire, à l'intérieur du dossier plugins.

```
/app
     /plugins
         /pizza
             /tests
                  /cases
                  /fixtures
                  /groups
```

Ils fonctionnent tout simplement comme des tests normaux, mais vous devez penser à utiliser les conventions de nommage pour les plugins lors de l'import des classes. Ceci est un exemple de cas de test pour le modèle CommandePizza vu au chapitre plugins de ce manuel. Une différence par rapport aux autres tests se trouve à la première ligne où 'Pizza.CommandePizza' est importée. Vous avez aussi besoin de préfixer les fixtures de votre plugin avec '`plugin.nom_plugin`'.

```
<?php
App::import('Model', 'Pizza.CommandePizza');

class CommandePizzaCase extends CakeTestCase {

    // Fixtures du plugin situées dans /app/plugins/pizza/tests/fixtures/
    var $fixtures = array('plugin.pizza.commande_pizza');
    var $CommandePizzaTest;

    function testerQuelqueChose() {
```

```
        // ClassRegistry indique au modèle d'utiliser la connexion à la base
de données de test
        $this->CommandePizzaTest =& ClassRegistry::init('CommandePizza');

        // effectuer quelque test utile ici
        $this->assertTrue(is_object($this->CommandePizzaTest));
    }
}
?>
```

Si vous voulez utiliser les fixtures de plugin dans les tests de votre application, vous pouvez les référencer en utilisant la syntaxe 'plugin.nomPlugin.nomFixture' dans le tableau $fixtures.

C'est tout ce qu'il y a à dire.

4.7.11 Divers

Le reporter de test standard est **très** minimaliste. Si vous voulez une sortie plus reluisante pour épater quelqu'un, ne vous inquiétez pas, il est très simple à étendre en fait.
Le seul danger c'est que vous devez bidouiller le code du coeur de Cake, plus particulièrement **/cake/tests/libs/cake_reporter.php**.

Pour changer la sortie test, vous pouvez surcharger les méthodes suivantes :

paintHeader()
: S'affiche avant le début du test.

paintPass()
: S'affiche chaque fois qu'un cas de test est réussi. Utilisez $this->getTestList() pour obtenir un tableau d'informations afférentes au test et $message pour obtenir le résultat du test. Pensez à appeler parent::paintPass($message).

paintFail()
: S'affiche chaque fois qu'un cas de test a échoué. Pensez à appeler parent::paintFail($message).

paintFooter()
: S'affiche quand le test est fini, i.e. quand tous les cas de tests ont été exécutées.

Si, quand paintPass et paintFail s'exécutent, vous voulez masquer la sortie parente, entourez l'appel par des balises de commentaires HTML, comme çà :

```
echo "\n<!-- ";
parent::paintFail($message);
echo " -->\n";
```

Voici un exemple de configuration de **cake_reporter.php** qui crée une table pour présenter les résultats du test :

```
<?php
 /**
 * CakePHP(tm) Tests <https://trac.cakephp.org/wiki/Developement/TestSuite>
 * Copyright 2005-2008, Cake Software Foundation, Inc.
 *                                1785 E. Sahara Avenue, Suite 490-204
 *                                Las Vegas, Nevada 89104
 *
 *  Licensed under The Open Group Test Suite License
 *  Redistributions of files must retain the above copyright notice.
```

```
 */
 class CakeHtmlReporter extends HtmlReporter {
 function CakeHtmlReporter($characterSet = 'UTF-8') {
 parent::HtmlReporter($characterSet);
 }

function paintHeader($testName) {
  $this->sendNoCacheHeaders();
  $baseUrl = BASE;
  print "<h2>$testName</h2>\n";
  print "<table
style=\"\"><th>Rés.</th><th>Cas de
Test</th><th>Message</th>\n";
  flush();
 }

 function paintFooter($testName) {
   $colour = ($this->getFailCount() + $this->getExceptionCount() > 0 ?
"red" : "green");
   print "</table>\n";
   print "<div style=\"";
   print "padding: 8px; margin-top: 1em; background-color: $colour; color:
white;";
   print "\">";
   print $this->getTestCaseProgress() . "/" . $this->getTestCaseCount();
   print " cas de test terminés :\n";
   print "<strong>" . $this->getPassCount() . "</strong>
réussites, ";
   print "<strong>" . $this->getFailCount() . "</strong>
échecs et ";
   print "<strong>" . $this->getExceptionCount() .
"</strong> exceptions.";
   print "</div>\n";
 }

 function paintPass($message) {
   parent::paintPass($message);
   echo "<tr>\n\t<td width=\"20\" style=\"border: dotted 1px;
border-top: hidden; border-left: hidden;                    border-right:
hidden\">\n";
   print "\t\t<span style=\"color:
green;\">Réussi</span>: \n";
   echo "\t</td>\n\t<td width=\"40%\" style=\"border: dotted
1px; border-top: hidden; border-left: hidden; border-right: hidden\">\n";
   $breadcrumb = $this->getTestList();
   array_shift($breadcrumb);
   array_shift($breadcrumb);
   print implode("-&gt;", $breadcrumb);
   echo "\n\t</td>\n\t<td width=\"40%\" style=\"border:
dotted 1px; border-top: hidden; border-left: hidden; border-right:
hidden\">\n";
   $message = split('at \[', $message);
   print "-&gt;$message[0]<br />\n\n";
   echo "\n\t</td>\n</tr>\n\n";
 }

 function paintFail($message) {
   echo "\n<!-- ";
   parent::paintFail($message);
   echo " -->\n";
   echo "<tr>\n\t<td width=\"20\" style=\"border: dotted 1px;
border-top: hidden; border-left: hidden; border-right: hidden\">\n";
   print "\t\t<span style=\"color: red;\">Raté</span>:
\n";
   echo "\n\t</td>\n\t<td width=\"40%\" style=\"border:
```

```
dotted 1px; border-top: hidden; border-left: hidden; border-right:
hidden\">\n";
   $breadcrumb = $this->getTestList();
   print implode("-&gt;", $breadcrumb);
   echo "\n\t</td>\n\t<td width=\"40%\" style=\"border:
dotted 1px; border-top: hidden; border-left: hidden; border-right:
hidden\">\n";
   print "$message";
   echo "\n\t</td>\n</tr>\n\n";
 }

 function _getCss() {
   return parent::_getCss() . ' .pass { color: green; }';
 }

 }
 ?>
```

4.7.11.2 Test Reporter methods

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Reporters have a number of methods used to generate the various parts of a Test suite response.

paintDocumentStart()
: Paints the start of the response from the test suite. Used to paint things like head elements in an html page.

paintTestMenu()
: Paints a menu of available test cases.

testCaseList()
: Retrieves and paints the list of tests cases.

groupCaseList()
: Retrieves and paints the list of group tests.

paintHeader()
: Prints before the test case/group test is started.

paintPass()
: Prints everytime a test case has passed. Use $this->getTestList() to get an array of information pertaining to the test, and $message to get the test result. Remember to call parent::paintPass($message).

paintFail()
: Prints everytime a test case has failed. Remember to call parent::paintFail($message).

paintSkip()
: Prints everytime a test case has been skipped. Remember to call parent::paintSkip($message).

paintException()
: Prints everytime there is an uncaught exception. Remember to call parent::paintException($message).

paintError()
: Prints everytime an error is raised. Remember to call parent::paintError($message).

paintFooter()
: Prints when the test case/group test is over, i.e. when all test cases has been executed.

paintDocumentEnd()
: Paints the end of the response from the test suite. Used to paint things like footer elements in an html page.

4.7.11.3 Grouper les tests

Si vous voulez faire fonctionner plusieurs de vos tests en même temps, vous pouvez essayer de créer un groupe de test. Créez un fichier dans **/app/tests/groups/** et nommez-le de cette façon **nom_votre_groupe_test.group.php**. Dans ce fichier, créez une classe qui étend **GroupTest** et importez les tests comme suit :

```
<?php
class EssaiGroupTest extends GroupTest {
  var $label = 'Essai';
  function essaiGroupTest() {
    TestManager::addTestCasesFromDirectory($this, APP_TEST_CASES . DS . 'models');
  }
}
?>
```

Le code ci-dessus groupera tout les cas de tests trouvés dans le dossier **/app/tests/cases/models/**. Pour ajouter un fichier individuel, utilisez **TestManager::addTestFile**($this, nom_fichier).

4.7.12 Lancer les tests depuis la ligne de commande

Si vous avez installé simpletest, vous pouvez lancer vos tests depuis la ligne de commande de votre application.

Depuis **app/**, exécutez :

```
cake testsuite help



Usage:
        cake testsuite category test_type file
                - category - "app", "core" or name of a plugin
                - test_type - "case", "group" or "all"
                - test_file - file name with folder prefix and without the (test|group).php suf

Examples:
                cake testsuite app all
                cake testsuite core all

                cake testsuite app case behaviors/debuggable
                cake testsuite app case models/my_model
                cake testsuite app case controllers/my_controller

                cake testsuite core case file
                cake testsuite core case router
                cake testsuite core case set

                cake testsuite app group mygroup
                cake testsuite core group acl
                cake testsuite core group socket

                cake testsuite bugs case models/bug
                  // for the plugin 'bugs' and its test case 'models/bug'
                cake testsuite bugs group bug
                  // for the plugin bugs and its test group 'bug'

Code Coverage Analysis:
```

```
Append 'cov' to any of the above in order to enable code coverage analysis
```

Comme le suggère le menu help, vous serez en mesure de lancer tous vos tests, une partie ou un seul cas de test depuis votre app, votre plugin ou le coeur, simplement depuis la ligne de commande.

Si vous avez un modèle de test dans **test/models/mon_modele.test.php**, vous lancerez simplement ce cas de test en exécutant :

```
cake testsuite app models/mon_modele
```

4.7.13 Test Suite changes in 1.3

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The TestSuite harness for 1.3 was heavily refactored and partially rebuilt. The number of constants and global functions have been greatly reduced. Also the number of classes used by the test suite has been reduced and refactored. You **must** update `app/webroot/test.php` to continue using the test suite. We hope that this will be the last time that a change is required to `app/webroot/test.php`.

Removed Constants

These constants have all been replaced with instance variables on the reporters and the ability to switch reporters.

Removed functions

These methods and the logic they contained have been refactored/rewritten into `CakeTestSuiteDispatcher` and the relevant reporter classes. This made the test suite more modular and easier to extend.

Removed Classes

- HtmlTestManager
- TextTestManager
- CliTestManager

These classes became obsolete as logic was consolidated into the reporter classes.

Modified methods/classes

The following methods have been changed as noted.

static. static. static. static. static. static.

testsuite Console changes

The output of errors, exceptions, and failures from the testsuite console tool have been updated to remove redundant information and increase readability of the messages. If you have other tools built upon the testsuite console, be sure to update those tools with the new formatting.

CodeCoverageManager changes

been moved to `CodeCoverageManager::init()` generation. collected coverage reports.

4.8 Internationalisation et Localisation

L'une des meilleures façons pour que vos applications touchent une audience plus large est de s'adresser à des langues multiples. Cela peut souvent s'avérer décourageant, mais les outils d'internationalisation et de localisation de CakePHP rendent cela plus facile.

Tout d'abord, il est important de comprendre la terminologie. *Internationalisation* désigne la capacité d'une application d'être localisée. Le terme *localisation* désigne l'adaptation d'une application pour qu'elle corresponde aux besoins d'une langue (ou d'une culture) spécifique (i.e., une "locale"). Internationalisation et localisation sont souvent abréviées respectivement en i18n et l10n ; 18 et 10 sont le nombre de caractères entre le premier et le dernier caractère.

4.8.1 Localiser votre Application

Il n'y a que quelques étapes à franchir pour passer d'une application uni-langue à une application multi-langue, dont la première est d'utiliser la fonction `__()` dans votre code. Ci-dessous, un exemple de code pour une application uni-langue :

```
<h2>Posts</h2>
```

Pour localiser votre code, tout ce que vous devez faire est d'inclure vos chaînes de caractères dans la fonction de traduction comme suit :

```
<h2><?php __('Posts') ?></h2>
```

Si vous ne faites 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 (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, 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 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 ne sont pas utilisés par CakePHP, ils sont les templates utilisés pour créer ou mettre à jour vos fichiers po, qui contiennent les traductions. Cake cherchera après vos fichiers po dans les dossiers suivants :

```
/app/locale/<locale>/LC_MESSAGES/<domaine>.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/fre/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 correcte 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 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 mais si vous créez des locales régionales (en_US, en_GB, etc.) Cake les utilisera dans les cas appropriés.

Il y a une limite de 1014 caractères pour chaque valeur *msgstr*.

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.
function beforeFilter() {
  $locale = Configure::read('Config.language');
  if ($locale && file_exists(VIEWS . $locale . DS . $this->viewPath)) {
    // utilise /app/views/fre/pages/tos.ctp au lieu de /app/views/pages/tos.ctp
    $this->viewPath = $locale . DS . $this->viewPath;
  }
}
```

ou

```
// Code de vue
echo $this->element(Configure::read('Config.language') . '/tos')
```

4.8.2 L'internationalisation dans CakePHP

Utiliser des données locales dans votre application est simple. La première étape consiste à préciser à CakePHP quelle langue l'application doit utiliser pour délivrer le contenu. Cela peut-être fait, entre-autres, en détectant les sous-domaines (en.exemple.com ou fra.exemple.com) ou en récupérant l'agent utilisateur du navigateur.

C'est mieux de faire le changement de langue dans le contrôleur :

```
$this->L10n = new L10n();
$this->L10n->get("fra");
```

Vous pourriez avoir envie de placer ce code dans le beforeFilter pour que chaque action du contrôleur soit affichée dans la langue correcte ou bien vous pourriez vouloir le placer dans une action qui manipule l'authentification ou le changement de langue par défaut.

L'affichage localisé d'un contenu se fait en utilisant la fonction pratique __(). Cette fonction est disponible globalement, mais elle sera probablement mieux utilisée dans vos vues. Le premier paramètre de la fonction

est le msgid défini dans les fichiers .po. Le contenu localisé est par défaut affiché (par un echo), mais un second paramètre optionnel entraîne à la place, le retour de la valeur (utile pour le surlignement ou les liens utilisant le TextHelper par exemple). Le court exemple ci-dessous montre comment afficher du contenu local en utilisant la fonction __(). Le rendu exact dépend de la locale qui a été sélectionnée en utilisant la classe L10n et de la définition active de la configuration.

```
<?php __("achat"); ?>
```

Souvenez-vous d'utiliser le paramètre d'échappement des différentes méthodes du helper si vous envisagez d'utiliser les données localisées comme une partie d'un appel de méthode du helper. Remarquez l'usage du second paramètre de __() pour retourner la donnée au lieu de la rendre par un echo () :

```
<?php
echo $form->error(
    'Carte.numeroCarte',
    __("erreurNumeroCarte", true),
    array('escape' => false)
);
?>
```

Si vous aimeriez avoir tous vos messages d'erreurs traduits par défaut, une solution simple serait d'ajouter le code suivant dans votre app_model.php :

```
function invalidate($field, $value = true) {
    return parent::invalidate($field, __($value, true));
}
```

4.9 Pagination

L'un des principaux obstacles dans la création d'une application web flexible et facile à utiliser est le design d'une interface utilisateur intuitive. Beaucoup d'applications ont rapidement tendance à grandir en taille et en complexité, et les designers et les programmeurs pensent qu'ils sont incapables de s'en sortir en montrant des centaines de milliers d'enregistrements. La refactorisation prend du temps, et les performances et la satisfaction de l'utilisateur peuvent en souffrir.

Afficher un nombre raisonnable d'enregistrements par page a toujours été une partie critique de toute application et a pour habitude de causer beaucoup de maux de tête aux développeurs. CakePHP diminue l'encombrement du développeur en fournissant une méthode rapide et simple pour organiser les informations. Le "PaginatorHelper" offre une excellente solution car il est simple à utiliser. Mis à part la pagination, il comprend quelques méthodes de tri également très simple d'utilisation. Le tri et la pagination avec Ajax est aussi supporté.

4.9.1 Configuration du contrôleur

Dans le contrôleur, nous commençons par définir la pagination par défaut dans la variable $paginate. Il est important de noter ici, que la clé *order* doit être définie dans une structure en tableau.

```
class RecettesController extends AppController {

    var $paginate = array(
        'limit' => 25,
```

```
        'order' => array(
            'Post.titre' => 'asc'
        )
    );
}
```

Vous pouvez aussi inclure d'autres options de find(), tel que *fields* :

```
class RecettesController extends AppController {

    var $paginate = array(
        'fields' => array('Post.id', 'Post.created'),
        'limit' => 25,
        'order' => array(
            'Post.titre' => 'asc'
        )
    );
}
```

Les autres clés qui peuvent être incluses dans le tableau *$paginate* sont similaires aux paramètres de la méthode *Model->find('all')*, c'est-à-dire :*conditions*, *fields*, *order*, *limit*, *page*, *contain* et *recursive*. En fait, vous pouvez définir plus d'une configuration de pagination dans le contrôleur, vous nommez simplement les parties du tableau, d'après le modèle que vous souhaitez configurer :

```
class RecettesController extends AppController {

    var $paginate = array(
        'Recette' => array (...),
        'Auteur' => array (...)
    );
}
```

Exemple de syntaxe utilisant le Comportement Containable :

```
class RecettesController extends AppController {

    var $paginate = array(
        'limit' => 25,
        'contain' => array('Article')
    );
}
```

Une fois la variable $paginate définie, nous pouvons appeler la méthode *paginate()* dans les actions du contrôleur. Cette méthode renvoie les résultats paginés d'un *find()* provenant du modèle et récupère quelques statistiques additionnelles sur la pagination, qui sont passées en arrière-plan à la vue. Cette méthode ajoute aussi l'assistant Paginator à la liste des assistants dans le contrôleur, s'il n'a pas déjà été ajouté.

```
function liste_recettes() {
    // similaire à findAll(), mais récupère les résultats sous
forme paginée
    $data = $this->paginate('Recette');
    $this->set('data', $data);
}
```

Vous pouvez filtrer les enregistrements en passant des conditions via le second paramètre de la fonction `paginate()` :

```
$data = $this->paginate('Recette', array('Recette.titre LIKE' => 'a%'));
```

Ou vous pouvez aussi configurer la clé *conditions* et d'autres clés dans le tableau `$paginate` à l'intérieur de votre action.

```
function liste_recettes() {
    $this->paginate = array(
        'conditions' => array('Recette.titre LIKE' => 'a%'),
        'limit' => 10
    );
    $data = $this->paginate('Recette');
    $this->set(compact('data'));
);
```

4.9.2 Pagination dans les vues

C'est à vous de décider comment afficher les enregistrements dans la vue, la plupart du temps, ils le seront dans des tables HTML. Les exemples suivants supposent une mise en page tabulaire, mais l'assistant de pagination disponible dans les vues n'a pas toujours besoin d'être construit de cette façon.

Voyez les détails dans la section PaginatorHelper de l'API.

Comme mentionné précédemment, l'assistant de pagination offre différentes options de tri qui peuvent être facilement intégrées dans les cellules d'en-tête de vos tableaux.

```
// app/views/recettes/liste_recettes.ctp
<table>
    <tr>
        <th><?php echo $this->Paginator->sort('ID', 'id'); ?></th>
        <th><?php echo $this->Paginator->sort('Titre', 'titre'); ?></th>

    </tr>
       <?php foreach($data as $recette): ?>
    <tr>
        <td><?php echo $recette['Recette']['id']; ?> </td>
        <td><?php echo $recette['Recette']['titre']; ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

Le lien créé, provenant de la méthode sort() de l'assistant de pagination, autorise l'utilisateur à cliquer sur les cellules d'en-tête pour changer le tri des données par le champ cliqué.

Il est aussi possible de trier une colonne basée sur les associations :

```
<table>
    <tr>
        <th><?php echo $this->Paginator->sort('Titre', 'titre'); ?></th>

        <th><?php echo $this->Paginator->sort('Auteur', 'Auteur.nom');
?></th>
    </tr>
       <?php foreach($data as $recette): ?>
    <tr>
        <td><?php echo $recette['Recette']['titre']; ?> </td>
        <td><?php echo $recette['Auteur']['nom']; ?> </td>
    </tr>
    <?php endforeach; ?>
</table>
```

La touche finale pour afficher la pagination dans les vues est l'addition des pages, également fournie par l'assistant de pagination

```
<!-- Affiche le nombre de pages -->
<?php echo $this->Paginator->numbers(); ?>
<!-- Affiche les liens des pages précédentes et suivantes -->
<?php
    echo $this->Paginator->prev('« Précédent ', null, null,
array('class' => 'disabled'));
    echo $this->Paginator->next(' Suivant »', null, null, array('class' =>
'disabled'));
?>
<!-- Affiche X de Y, où X est la page courante et Y le nombre de pages -->
<?php echo $this->Paginator->counter(); ?>
```

Le résultat de la méthode counter() peut-être personnalisé grâce à des marqueurs spécifiques

```
<?php
echo $this->Paginator->counter(array(
    'format' => 'Page %page% de %pages%, montrant %current% enregistrements sur un total
de %count%, en commençant à %start% et se terminant à %end%'
));
?>
```

Pour passer toute l'URL en paramètre de la fonction de pagination, faites comme suit :

```
    $this->Paginator->options(array('url' => $this->passedArgs));
```

Pour faire passer des éléments en paramètre non-standard, vous devrez les fusionner manuellement avec `$this->passedArgs` :

```
// pour les urls comme http://www.exemple.com/fr/controller/action
// qui sont routées par : Router::connect('/:lang/:controller/:action/*',
array(),array('lang'=>'fr|en'));
$this->Paginator->options(array('url'=>array_merge(array('lang'=>$lang),$this->passedArgs)));
```

Ou bien vous pouvez spécifier quels paramètres passer manuellement :

```
    $this->Paginator->options(array('url' =>  array("0", "1")));
```

4.9.3 Pagination AJAX

Par défaut en 1.3 le <code>PaginatorHelper</code> utilise JsHelper pour les fonctionnalités ajax. Cependant, si vous ne voulez pas cela et que vous voulez utiliser <code>Ajaxhelper</code> ou un Helper personnalisé pour les liens ajax, vous pouvez le faire en changeant le <code>$helpers</code> tableau array dans votre contrôleur. Après le démarrage de <code>paginate()</code> faire la chose suivante. <pre> $this->set('posts', $this->paginate()); $this->helpers['Paginator'] = array('ajax' => 'Ajax'); </pre> Changer <code>PaginatorHelper</code> pour utiliser le <code>AjaxHelper</code> pour les opérations ajax. Vous pouvez également définir la clé 'ajax' à tous les helper, aussi longtemps que cette class implémente une méthode <code>link()</code> qui se comporte comme <code>HtmlHelper::link()</code>

4.9.4 Requête de pagination personnalisée

Fix me: Please add an example where overriding paginate is justified

Un bon exemple de cas où vous auriez besoin de ceci, c'est si la base de données sous-jacente ne supporte pas la syntaxe SQL LIMIT. Ceci est vrai pour DB2 d'IBM. Vous pouvez quand même utiliser la pagination CakePHP en ajoutant la requête personnalisée au modèle.

Si vous devez créer des requêtes personnalisées pour générer les données que vous souhaitez paginer, vous pouvez surcharger les méthodes du modèle `paginate()` et `paginateCount()` utilisées par la logique de pagination du contrôleur.

Avant de continuer, vérifiez que vous ne pouvez pas atteindre votre objectif avec les méthodes du modèle de base.

La méthode `paginate()` utilise les mêmes paramètres que `Model::find()`. Pour utiliser votre propre méthode/logique, surchargez-là dans le modèle souhaité pour obtenir les données.

```
/**
 * Surcharge de la méthode paginate() - grouper par semaine, equipe_exterieure_id
et equipe_locale_id
 */
function paginate($conditions, $fields, $order, $limit, $page = 1, $recursive = null,
$extra = array()) {
    $recursive = -1;
    $group = $fields = array('semaine', 'equipe_exterieure_id', 'equipe_locale_id');
     return $this->find('all', compact('conditions', 'fields', 'order', 'limit', 'page',
'recursive', 'group'));
}
```

Vous devez aussi surcharger le `paginateCount()` du coeur, cette méthode attend les mêmes paramètres que `Model::find('count')`. L'exemple suivant utilise quelques fonctions spécifiques à Postgresql, merci de bien vouloir l'ajuster en fonction de la base de données que vous utilisez.

```
/**
```

```
 * Surcharge de la méthode paginateCount()
 */
function paginateCount($conditions = null, $recursive = 0, $extra = array()) {
    $sql = "SELECT DISTINCT ON(semaine, equipe_locale_id, equipe_exterieure_id) semaine,
equipe_locale_id, equipe_exterieure_id FROM parties";
    $this->recursive = $recursive;
    $results = $this->query($sql);
    return count($results);
}
```

Le lecteur attentif aura remarqué que la méthode de pagination que nous avons définie n'était pas vraiment nécessaire. Tout ce que nous avons à faire est d'ajouter le mot-clé dans la variable de classe `$paginate` du contrôleur.

```
/**
* Ajout de la clause GROUP BY
*/
var $paginate = array(
    'MonModele' => array('limit' => 20,
                         'order' => array('semaine' => 'desc'),
                         'group' => array('semaine', 'equipe_locale_id',
'equipe_exterieure_id'))
                        );

/**
* Ou bien à la volée, dans l'action du contrôleur
*/
function index() {
    $this->paginate = array(
        'MonModele' => array('limit' => 20,
                         'order' => array('semaine' => 'desc'),
                         'group' => array('semaine', 'equipe_locale_id',
'equipe_exterieure_id'))
                        );
```

Cependant, il sera tout de même nécessaire de surcharger la méthode `paginateCount()` pour obtenir une valeur exacte.

4.10 REST

De nombreux programmeurs commencent à se rendre compte de la nécessité de donner accès au coeur de leur application à un public plus large. Fournir un accès direct et aisé au coeur de votre API peut aider votre plateforme à être acceptée, permet la création d'applications composites (*mashups*) et une intégration simple avec d'autres systèmes.

Bien que d'autres solutions existent, REST est une bonne manière de fournir un accès facile à la logique que vous avez créée dans votre application. C'est simple, généralement basé sur du XML (nous parlons de XML simple, rien de comparable à une enveloppe SOAP), et repose sur les entêtes HTTP pour la définition des actions à effectuer. Exposer une API via REST avec CakePHP est simple.

4.10.1 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 Routeur (*Router*) comporte une méthode appelée mapResources() utilisée pour mettre en place un certain nombre de routes par défaut accédant par REST à vos contrôleurs. 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('recettes');
Router::parseExtensions();
```

La première ligne met en place un certain nombre de routes par défaut pour un accès facile par REST. Ces routes correspondent aux méthodes de requêtes HTTP.

Méthode HTTP	URL	Action de contrôleur appelée
GET	/recettes	RecettesController::index()
GET	/recettes/123	RecettesController::view(123)
POST	/recettes	RecettesController::add()
PUT	/recettes/123	RecettesController::edit(123)
DELETE	/recettes/123	RecettesController::delete(123)
POST	/recettes/123	RecettesController::edit(123)

La classe Routeur (*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. Le X_HTTP_METHOD_OVERRIDE
3. L'entête REQUEST_METHOD

La méthode POST *_method* 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 routeur est paramétré pour faire correspondre les requêtes REST à certaines actions de contrôleur, nous pouvons nous mettre à créer la logique dans nos actions de contrôleur. Un contrôleur basique pourrait ressembler à ceci :

```
// controllers/recettes_controller.php

class RecettesController extends AppController {

    var $components = array('RequestHandler');

    function index() {
        $recettes = $this->Recette->find('all');
        $this->set(compact('recettes'));
    }

    function view($id) {
        $recette = $this->Recette->findById($id);
        $this->set(compact('recette'));
    }

    function edit($id) {
        $this->Recette->id = $id;
```

```
        if ($this->Recette->save($this->data)) {
            $message = 'Sauvegardé';
        } else {
            $message = 'Erreur';
        }
        $this->set(compact("message"));
      }

    function delete($id) {
        if($this->Recette->del($id)) {
            $message = 'Supprimé';
        } else {
            $message = 'Erreur';
        }
        $this->set(compact("message"));
    }
}
```

Comme nous avons ajouté un appel à Router::parseExtensions(), le routeur de CakePHP est déjà prêt à servir différentes vues selon différents types de requêtes. Comme nous avons affaire à des requêtes REST, le type de la vue est du XML. Nous plaçons les vues REST pour notre RecettesController dans app/views/xml. Nous pouvons également utiliser le XMLHelper pour faciliter l'affichage du XML dans ces vues. Voici ce à quoi pourrait ressembler notre vue d'index :

```
// app/views/recettes/xml/index.ctp

<recettes>
    <?php echo $xml->serialize($recettes); ?>
</recettes>
```

Les utilisateurs expérimentés de CakePHP auront peut-être remarqué que nous n'avons pas inclus le XMLHelper dans le tableau $helpers de notre RecettesController. C'est fait exprès - lorsque nous servons un type de contenu spécifique grâce à parseExtensions(), CakePHP recherche automatiquement un Assistant (*Helper*) correspondant au type. Comme nous utilisons XML comme type de contenu, le XMLHelper est automatiquement chargé pour être utilisé dans ces vues.

Le XML généré ressemblera à quelque chose comme ceci :

```
<articles>
    <article id="234" created="2008-06-13" modified="2008-06-14">
        <auteur id="23423" first_name="Billy" last_name="Bob"></author>
        <commentaire id="245" body="Un commentaire pour cet article."></comment>
    </article>
    <article id="3247" created="2008-06-15" modified="2008-06-15">
        <auteur id="625" first_name="Nate" last_name="Johnson"></author>
        <commentaire id="654" body="Un commentaire pour cet article."></comment>
    </article>
</articles>
```

Créer la logique pour l'action *edit* est un plus complexe, mais de peu. Comme vous fournissez une API renvoyant du XML, c'est un choix naturel que de choisir du XML comme format d'entrée. Pas d'inquiétude cependant : les classes RequestHandler et Router facilitent grandement les choses. Si une requête POST ou PUT a un type de contenu XML, alors les données d'entrée sont passées à une instance de l'objet XML de Cake qui est assigné à la propriété $data du contrôleur. Grâce à cette fonctionnalité, manipuler des données XML et POST en parallèle est transparent : aucun changement n'est nécessaire dans le code du contrôleur ou

du modèle. Tout ce dont vous avez besoin devrait être retrouvé dans $this->data.

4.10.2 Routage REST personnalisé

Si les routes par défaut créées par mapResources() ne vous conviennent pas, 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 pour une URL donnée. Le premier paramètre est l'URL elle-même, le deuxième vous permet de fournir ces options. Le troisième paramètre vous permet de spécifier des motifs (*patterns*) d'expressions régulières pour aider CakePHP à identifier certains marqueurs dans l'URL spécifiée.

Nous allons fournir ici un exemple simple et vous permettre d'adpter cette route pour vos autres actions *RESTful*. Voici ce à quoi ressemblerait notre route REST *edit*, sans utiliser mapResources() :

```
Router::connect(
    "/:controller/:id",
    array("action" => "edit", "[method]" => "PUT"),
    array("id" => "[0-9+]")
)
```

Les techniques avancées de routage sont décrites ailleurs, nous allons donc nous concentrer sur le point le plus important pour notre but ici : la clé *method* du tableau options dans le deuxième paramètre. Une fois cette clé affectée, la route spécifiée fonctionne uniquement pour cette méthode de requête HTTP (qui pourrait également être GET, DELETE, etc).

5 Composants intégrés

CakePHP contient un certain nombre de composants intégrés. Ils fournissent des fonctionnalités toutes prêtes pour de nombreuses tâches couramment utilisées.

Acl	Le composant Acl fournit une interface facile à utiliser pour les listes de contrôles d'accès basées sur une base de données ou un fichier ini.
Auth	Le composant Auth fournit un système d'authentification facile à utiliser, à travers une grande variété de processus d'authentification, comme les *callbacks* de contrôleur, l'Acl ou les *callbacks* du modèle Object.
Cookie	Le composant Cookie se comporte d'une façon similaire au composant Session, dans le sens où il fournit une encapsulation pour le support natif des cookies en PHP.
Email	Une interface qui peut être utilisée pour envoyer des emails grâce à l'un des nombreux agents de transfert de mail existant, y compris la fonction mail() de php et le smtp.
RequestHandler	Le RequestHandler vous permet d'analyser plus finement les requêtes de vos visiteurs et de renseigner votre application sur les types de contenus et les informations demandés.
Security	Le composant Security vous permer de définir une sécurité renforcé, d'utiliser et de managee l'authentification HTTP.
Session	Le composant Session fournit un gestionnaire de stockage indépendant pour les sessions PHP.

Pour en savoir plus à propos de chaque composant, voyez le menu sur la gauche ou apprenez comment créer vos propres composants.

Tous les composants du coeur peuvent maintenant être configuré dans le tableau `$components` d'un contrôleur.

```
<?php
class AppController extends Controller {

    var $components = array(
        'Auth' => array(
            'loginAction' => array('controller' => 'users', 'action' =>
'inscription'),
            'fields' => array('username' => 'email', 'password' =>
'mot_de_passe'),
        ),
        'Security',
        'Email' => array(
            'from' => 'webmaster@domaine.com',
            'sendAs' => 'html',
        ),
    );
}
?>
```

Vous pouvez surcharger les paramètres dans le `beforeFilter()` du contrôleur.

```
<?php
class MembresController extends AppController {
```

```
    function beforeFilter() {
        $this->Email->from = 'support@domaine.com';
    }
}
?>
```

5.1 Listes de Contrôle d'Accès (ACL)

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.

5.1.1 Comprendre comment les ACL fonctionnent

Les choses puissantes 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 contrôleurs 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 contrôleur 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 recherchée
- 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

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 :

	Les armes	**L'Anneau**	**Le porc salé**	**La diplomatie**	**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

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(tm)
  - ♦ 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.

Noeud de l'ARO	Information sur la permission	Résultat
La Communauté de l'Anneau	Refuse tout	Refuser l'accès à la bière.
Les Hobbits	Autorise la bière	Autoriser l'accès à la bière !
Pippin	--	Toujours autoriser la bière !

Puisque le noeud "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 :

Noeud 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

5.1.2 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 utilisé 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 qui 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 :

```
;-------------------------------------
; Les 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 = biere

[pippin]
groups = hobbits

[gollum]
groups = visiteurs

;-------------------------------------
; Groupe de l'ARO
;-------------------------------------
```

```
[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 utilisant le composant ACL.

5.1.3 Définir les permissions : ACL de Cake via une base de données

L'implémentation pas 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 modèles du coeur et d'une application en mode console qui sont créés lors de votre installation de Cake. Les modèles sont utilisés par Cake pour interagir avec votre base de données, afin de stocker et de retrouver les noeuds 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 sÃ»r que votre `/app/config/database.php` soit 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.
```

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 ça 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).

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 composant ACL utilise le comportement en arbre de CakePHP pour gérer les héritages d'arbres. Les fichiers de modèle de classe pour ACL sont compilés dans un seul fichier db_acl.php.

Maintenant que nous avons tout configuré, attelons-nous à la création de quelques arbres ARO et ACO.

5.1.3.2 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 modèle 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.

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 modèle/clé étrangère.

Vous voulez créer de nouveaux objets ACL en utilisant le modèle ACL du coeur de CalePHP. 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 modèle (s'il en est). Par exemple, un certain nombre d'AROs correspondraient aux enregistrement 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 User avec un seul appel find() au modèle 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 modèle.

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 modèle. 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 et d'inclure les composants ACL de Cake dans votre tableau $composents du contrôleur :

```
var $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 contrôleur :

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 groups d'ARO. De fait que 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 contrôleur 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 ARIs et des ACOs à la volée.

Ce ne devrait rien avoir de radicalement nouveau - nous sommes justes entrain d'utiliser les modèles pour enregistrer les données comme nous le faisons toujours :

```
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 consile 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 modèle de notre base de données.

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).

```
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
modèle 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 ...
}
```

Typiquement vous n'aurez pas à fournir et un alias, et un modèle/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. Tant que nous pouvons structurer plus que par une représentation abstraite que celle de nos ACO, il est parfois plus pratique de modéliser un arbre ACO après que la configuration faite par le Contrôleur/Action de Cake. Nous avons cinq principaux objets à manipuler dans le scénario de la Communauté, pour la configuration naturelle de ce dernier dans une application Cake est un groupe de modèles, et enfin pour les contrôleurs qui le manipulent. A côté des contrôleurs eux-mêmes, nous allon vouloir contrôler l'accès à des actions spécifiques de ces contrôleurs.

Basés sur 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 fils 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 comment assigner les permissions.

Nous sommes maintenant des pro 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 modèle Aco du coeur.

5.1.3.3 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 composant Acl du coeur de CakePHP. Continuons avec notre exemple.

Ici nous travaillerons dans un contexte d'une action de contrôleur. Nous faisons cela parce que les permissions sont managées par le composant Acl.

```
class ChosesController extends AppController
{
    // Vous pourriez placer çà dans AppController
,
    // mais cela fonctionne bien ici aussi.

    var $components = array('Acl');

}
```

Configurons quelques permissions de base, en utilisant le Composant Acl dans une action à l'intérieur de ce contrôleur.

```
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('guerriers/Legolas', 'Armes', 'delete');
    $this->Acl->deny('guerriers/Gimli',   'Armes', 'delete');

    die(print_r('done', 1));
}
```

Le premier appel que nous faisons au composant 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 de prendre une décision un peu plus précise sur 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 modèle/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');
```

L'adressage d'un noeud en utilisant la syntaxe avec un alias, nécessite une chaîne délimitée par des slashs ('/utilisateurs/salaries/developpeurs'). L'adressage d'un noeud en utilisant la syntaxe modèle/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 composant Acl pour contrôler les permissions que nous venons de définir.

5.1.3.4 Vérification des Permissions : le Composant ACL

Utilisons le Composant Acl pour s'assurer que les nains et les elfes ne peuvent déplacer des choses depuis l'armurerie. Maintenant, nous devrions être en mesure d'utiliser le Composant 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 contrôleur :

```
function index()
{
    // Tout cela renvoie "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 modèle/clé
étrangère
    // pour nos AROs utilisateur
    $this->Acl->check(array('model' => 'User', 'foreign_key' => 2356), 'Armes');

    // Ceci retourne "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.

5.2 Authentification

Les systèmes d'authentification utilisateur constituent une partie courante de nombreuses applications Web. Dans CakePHP, il y a plusieurs systèmes pour authentifier des utilisateurs, chacun fournissant des options différentes. Au coeur du composant d'authentification, on vérifie si l'utilisateur possède un compte sur le site. Si c'est le cas, le composant donnera accès au site complet à l'utilisateur.

Ce composant peut être combiné avec le composant ACL (access control lists) pour créer des niveaux d'accès plus complexes à l'intérieur d'un site. Le composant ACL pourrait par exemple, vous permettre d'autoriser un utilisateur à accéder aux zones publiques du site, tout en autorisant un autre utilisateur d'accéder aux portions administratives protégées du site.

Le composant AuthComponent de CakePHP peut servir à créer un tel système rapidement et facilement. Regardons comment vous construiriez un système d'authentification vraiment simple.

Comme n'importe quel composants, vous l'utilisez en ajoutant 'Auth' à la liste des composants de votre contrôleur :

```
class FooController extends AppController {
    var $components = array('Auth');
```

Ou ajoutez-le à votre AppController, ainsi tous vos contrôleurs l'utiliseront :

```
class AppController extends Controller {
    var $components = array('Auth');
```

Maintenant, il y a quelques conventions auxquelles penser quand on utilise le composant AuthComponent. Par défaut, AuthComponent s'attend à ce que vous ayez une table appelée 'users' avec des champs nommées 'username' et 'password'.

Dans certaines situations, les bases de données ne vous laissent pas utiliser 'password' comme nom de colonne. Voyez Définir les variables du composant Auth, pour trouver un exemple sur la façon de changer le nom par défaut des champs afin de travailler avec votre propre environnement.

Définissons notre table users en utilisant le SQL suivant :

```
CREATE TABLE users (
    id integer auto_increment,
    username char(50),
    password char(50),
    PRIMARY KEY (id)
);
```

Quelque chose à garder à l'esprit lors de la création d'une table pour stocker toutes vos données d'authentification d'un utilisateur, c'est que le composant Auth s'attend à ce que la valeur du mot de passe stocké dans la base de données soit hashée, plutôt que d'être stockée en texte brut. Assurez-vous que le champ que vous utiliserez pour stocker les mots de passe est assez long pour stocker le hash (40 caractères pour SHA1, par exemple).

Si vous voulez ajouter un utilisateur manuellement dans la base de données, la méthode la plus simple pour obtenir la bonne donnée est de tenter de s'identifier et de regarder le log sql.

Pour le paramétrage le plus basique, il vous faudra seulement créer deux actions dans votre contrôleur :

```
class UsersController extends AppController {

    var $name = 'Users';
    var $components = array('Auth'); // Pas nécessaire si déclaré
dans votre contrôleur app

    /**
    * Le Composant Auth fournit la fonctionnalité nécessaire
    * pour le login, donc vous pouvez laisser cette fonction vide.
    */
    function login() {
    }
    function logout() {
        $this->redirect($this->Auth->logout());
    }
}
```

Bien que vous puissiez laisser la fonction login() vide, vous avez besoin de créer la template de vue login (sauvegardée dans app/views/users/login.ctp). C'est toutefois la seule template de vue du contrôleur Users que vous avez besoin de créer. L'exemple ci-dessous présuppose que vous utilisez l'assistant Form :

```
<?php
    echo $session->flash('auth');
    echo $form->create('User', array('action' => 'login'));
    echo $form->input('username');
    echo $form->input('password');
    echo $form->end('Login');
?>
```

Cette vue crée un formulaire d'identification où vous entrez un nom d'utilisateur et un mot de passe. Une fois que vous soumettez ce formulaire, le composant Auth s'occupe du reste pour vous. Le message de session flash affichera les notices générées par AuthComponent. Dès que l'identification est réussie, l'enregistrement de la base de données correspondant à l'utilisateur actuellement identifié est sauvegardé en session.

Croyez le ou non, on en a terminé ! Voici comment écrire un système d'authentification incroyablement simple, fondé sur la base de données, avec le composant Auth. Cependant, il y a encore plus de choses que nous pouvons faire. Intéressons-nous maintenant à quelques usages avancés du composant.

5.2.1 Configurer les variables du composant Auth

Chaque fois que vous voulez modifier une option par défaut du composant Auth, vous devez le faire en créant une méthode beforeFilter() dans votre contrôleur, puis en appelant les différentes méthodes pré-existantes ou en configurant les variables du composant.

Par exemple, pour changer le nom du champ utilisé pour le mot de passe de 'password' à 'mot_secret', vous devez faire ceci :

```
class UsersController extends AppController {
    var $components = array('Auth');

    function beforeFilter() {
        $this->Auth->fields = array(
            'username' => 'username',
            'password' => 'mot_secret'
            );
    }
}
```

Dans cette situation particulière, vous devrez aussi penser à changer le nom du champ dans la vue correspondante !

Une autre utilisation commune des variables du composant Auth est d'autoriser l'accès à certaines méthodes sans que l'utilisateur ne soit identifié (par défaut, Auth interdit l'accès à toutes les actions sauf aux méthodes login et logout).

Par exemple, si nous voulions autoriser tous les utilisateurs à accéder aux méthodes index et voir (mais à aucune autre), nous ferions comme çà :

```
function beforeFilter() {
        $this->Auth->allow('index','voir');
}
```

5.2.2 Afficher les messages d'erreur du composant Auth

Pour afficher les messages d'erreur que Auth renvoie, vous devez ajouter le code suivant à votre vue. Dans ce cas, le message apparaîtra à la suite des messages flash normaux.

Pour voir tous les messages flash normaux et les messages flash auth dans toutes les vues, ajoutez les deux lignes suivantes au fichier views/layouts/default.ctp, dans la section body, de préférence avant la ligne content_for_layout.

```
<?php
    $session->flash();
    $session->flash('auth');
?>
```

Pour personnaliser les messages d'erreur de Auth, placez le code suivant dans le contrôleur AppController ou partout où vous avez placé les paramètres de Auth.

```
<?php
$this->Auth->loginError = "Ce message apparaît lorsque les informations
d'identifications sont mauvaises ";

$this->Auth->authError = "Cette erreur se présente quand l'utilisateur tente
d'accéder à une partie du site qui est protégée";
?>
```

5.2.3 Diagnostic des problèmes avec Auth

Il peut être parfois un peu difficile de diagnostiquer les problèmes quand ça ne marche pas comme prévu, voici donc quelques points à se rappeler.

Hâchage du mot de passe

Quand vous postez des informations à une action via un formulaire, le composant Auth hâche (crypte) automatiquement le contenu de votre champ mot de passe, si vous avez également une donnée dans le champ 'username'. Donc, si vous essayez de créer une page d'inscription quelconque, assurez-vous que l'utilisateur ait rempli le champ 'confirmation du mot de passe' pour comparer les deux. Voici un exemple de code :

```
<?php
function enregistrer() {
    if ($this->data) {
        if ($this->data['User']['password'] ==
$this->Auth->password($this->data['User']['password_confirm'])) {
            $this->User->create();
            $this->User->save($this->data);
        }
    }
}
?>
```

5.2.3.1 Hashage du mot de passe

Le hashage automatique de votre champ mot de passe se produit **seulement** si les données postées contiennent à la fois les champs 'username' et 'password'.

Quand vous postez des informations à une action via un formulaire, le composant Auth hashe (crypte) automatiquement le contenu de votre champ mot de passe, si les données postées contiennent aussi le champ 'username'. Donc, si vous essayez de créer une page d'inscription quelconque, assurez-vous que l'utilisateur ait rempli un champ 'confirmation du mot de passe' pour comparer les deux. Voici un exemple de code :

```
<?php
function enregistrer() {
    if ($this->data) {
        if ($this->data['User']['password'] ==
$this->Auth->password($this->data['User']['password_confirm'])) {
            $this->User->create();
            $this->User->save($this->data);
        }
    }
}
?>
```

5.2.4 Changer la fonction de hâchage

Le composant Auth utilise la classe Security pour hacher un mot de passe. La classe Security utilise le procédé SHA1 par défaut. Pour changer la fonction de hash utilisée par le composant Auth, servez-vous de la méthode `setHash` en lui passant `md5`, `sha1` ou `sha256` comme premier et unique paramètre.

```
Security::setHash('md5'); // ou sha1 ou sha256.
```

La classe Security utilise une valeur *salt* (définie dans **/app/config/core.php**) pour hacher le mot de passe.

Si vous voulez utiliser une logique de hachage du mot de passe différente, autre que md5/sha1 ajouté au *salt* de l'application, vous devrez surcharger le mécanisme standard de hashPassword. Vous aurez besoin de faire cela si vous avez, par exemple, une base de données existante, qui utilisait précédemment un procédé de hachage sans *salt*. Pour faire cela, créez la méthode `hashPasswords` dans la classe à laquelle vous souhaitez confier le hachage de vos mots de passe (habituellement le modèle User) et définissez `authenticate` par l'objet sur lequel vous réalisez l'authentification (habituellement, c'est User), comme ceci :

```
function beforeFilter() {
   $this->Auth->authenticate = ClassRegistry::init('User');
   ...
   parent::beforeFilter();
}
```

Avec le code ci-dessus, la méthode hashPasswords() du modèle User sera appelée chaque fois que Cake appelle AuthComponent::hashPasswords().

5.2.5 Les Méthodes du composant Auth

action (string $action = ':controller/:action')

Si vous utilisez les ACOs dans le cadre de votre structure ACL, vous pouvez obtenir le chemin jusqu'au noeud ACO relié à un couple contrôleur/action particulier :

```
    $acoNode = $this->Auth->action('users/delete');
```

Si vous ne passez pas de valeur, le couple contrôleur/action courant est utilisé.

5.2.5.2 allow

Si vous avez des actions dans votre contrôleur que vous n'avez pas besoin d'authentifier (comme une action d'enregistrement d'un utilisateur), vous pouvez ajouter des méthodes que le composant Auth devrait ignorer. L'exemple suivant montre comment autoriser une action intitulée 'enregistrer'.

```
function beforeFilter() {
    ...
    $this->Auth->allow('enregistrer');
}
```

Si vous souhaitez autoriser plusieurs actions qui échapperont à l'authentification, passez-les en paramètres à la méthode allow() :

```
function beforeFilter() {
    ...
```

```
    $this->Auth->allow('foo', 'bar', 'baz');
}
```

Raccourci : vous pouvez aussi autoriser toutes les actions d'un contrôleur en utilisant '*'.

```
function beforeFilter() {
    ...
    $this->Auth->allow('*');
}
```

Si vous utilisez requestAction dans votre layout ou vos éléments, vous devriez autoriser ces actions de façon à être capable d'ouvrir la page de login proprement.

Le composant Auth suppose que les noms de vos actions respectent les conventions et qu'elles sont "underscorées".

5.2.5.3 deny

Il peut arriver que vous vouliez retirer des actions de la liste des actions autorisées (déclarée en utilisant $this->Auth->allow()). Voici un exemple :

```
    function beforeFilter() {
        $this->Auth->authorize = 'controller';
        $this->Auth->allow('delete');
    }

    function isAuthorized() {
        if ($this->Auth->user('role') != 'admin') {
            $this->Auth->deny('delete');
        }

        ...
    }
```

5.2.5.4 hashPasswords

`hashPasswords ($data)`

Cette méthode vérifie si `$data` contient les champs *username* et *password*, comme spécifié par la variable `$fields`, elle même indexée par le nom du modèle, comme spécifié dans `$userModel`. Si le tableau `$data` contient à la fois *username* et *password*, la méthode encode le champ *password* du tableau et retourne le tableau `$data` dans le même format. Cette fonction devrait être utilisée en priorité pour les requêtes d'insertion ou de mise à jour de l'utilisateur, quand le champ *password* est affecté.

```
    $data['User']['username'] = 'moi@moi.com';
    $data['User']['password'] = 'changemoi';
    $hashedPasswords = $this->Auth->hashPasswords($data);
    print_r($hashedPasswords);
    /* retourne :
    Array
    (
        [User] => Array
        (
```

```
            [username] => moi@moi.com
            [password] => 8ed3b7e8ced419a679a7df93eff22fae
        )
    )

    */
```

Le champ *$hashedPasswords['User']['password']* sera maintenant encodé en utilisant la fonction `password` du composant.

Si votre contrôleur utilise le composant Auth et que les données postées contiennent les champs mentionnés ci-dessus, il encodera automatiquement le mot de passe en utilisant cette fonction.

5.2.5.5 mapActions

Si vous utilisez les Acl en mode CRUD, vous aimeriez peut-être assigner certaines actions non-standards à chaque partie du CRUD.

```
$this->Auth->mapActions(
    array(
        'create' => array('uneAction'),
        'read' => array('uneAction', 'uneAction2'),
        'update' => array('uneAction'),
        'delete' => array('uneAction')
    )
);
```

5.2.5.6 login

`login($data = null)`

Si vous souhaitez une authentification depuis un composant Ajax, vous pouvez utiliser cette méthode pour authentifier manuellement un utilisateur dans le système. Si vous ne passez aucune valeur pour `$data`, les données reçues en POST seront alors automatiquement passées au controlleur.

5.2.5.7 logout

Cette méthode fournit une manière rapide de désauthentifier quelqu'un et de le rediriger là où il a besoin d'aller.

Cette méthode est également pratique si vous voulez proposer un lien 'Déconnexion' dans la partie membres de votre application.

Exemple :

```
$this->redirect($this->Auth->logout());
```

5.2.5.8 password

`password (string $password)`

Passez une chaîne à cette méthode et vous pourrez voir à quoi ressemblera le mot de passe crypté. C'est une

fonctionnalité essentielle si vous créez un écran d'inscription où les utilisateurs doivent entrer deux fois leur mot de passe pour le confirmer.

```
if ($this->data['User']['password'] ==
    $this->Auth->password($this->data['User']['password2'])) {

    // Les mots de passe correspondent, on continue
    ...
} else {
    $this->flash('Les mots de passe saisis ne correspondent pas', 'users/register');
}
```

Le composant Auth encryptera automatiquement le champ `password` si le champ `username` est aussi présent dans les données envoyées.

Cake ajoute un "grain de sécurité" (`Security.salt`) à votre chaîne de mot de passe et crypte le tout ensuite. La fonction de cryptage utilisée dépend de celle définie dans la classe utilitaire `Security` de CakePHP (sha1 par défaut). Vous pouvez utiliser la fonction `Security::setHash` pour changer la méthode de cryptage. Le "grain de sécurité" est configuré dans le fichier `core.php` de votre application.

5.2.5.9 user

`user(string $key = null)`

Cette méthode fournit des informations sur l'utilisateur connecté. Ces informations sont issues de la session. Par exemple :

```
if ($this->Auth->user('role') == 'admin') {
    $this->flash('Vous avez un accès administrateur');
}
```

Elle peut aussi être utilisée pour obtenir des informations complètes sur la session de l'utilisateur, de cette façon :

```
$data['User'] = $this->Auth->user();
```

Si cette méthode renvoie null, l'utilisateur n'est pas connecté.

Dans les vues, vous pouvez utiliser l'assistant Session, pour retrouver les informations sur l'utilisateur actuellement connecté :

```
$session->read('Auth.User'); // renvoie l'ensemble des informations sur l'utilisateur
$session->read('Auth.User.first_name') // renvoie la valeur d'un champ en particulier
```

La clé de session peut être différente en fonction du modèle configuré pour utiliser Auth. Par exemple, si vous utilisez le modèle `Compte` au lieu de `User`, alors la clé de session sera `Auth.Compte`.

5.2.6 Variables du composant Auth

Vous ne voulez pas utiliser un modèle Utilisateur pour vous authentifier ? Pas de problème, modifiez ce comportement en configurant cette variable avec le nom du modèle que vous voulez utiliser.

```
<?php
    $this->Auth->userModel = 'Membre';
?>
```

5.2.6.2 fields

Pour outrepasser les champs utilisateur et mot de passe utilisés par défaut pour l'authentification.

```
<?php
    $this->Auth->fields = array('username' => 'email', 'password' =>
'motdepasse');
?>
```

5.2.6.3 userScope

Utilisez cette propriété pour ajouter des contraintes supplémentaires afin que l'authentification réussisse.

```
<?php
    $this->Auth->userScope = array('Utilisateur.actif' => true);
?>
```

5.2.6.4 loginAction

Vous pouvez changer l'adresse de connexion par défaut */users/login* par toute action de votre choix.

```
<?php
    $this->Auth->loginAction = array('admin' => false, 'controller' =>
'membres', 'action' => 'login');
?>
```

5.2.6.5 loginRedirect

Le Composant Auth mémorise quelle paire contrôleur/action vous essayiez d'obtenir avant que l'on vous demande de vous authentifier, en stockant cette valeur dans la Session, sous la clé **`Auth.redirect`**. Cependant, si cette valeur de session n'est pas définie (par exemple, si vous arrivez à la page d'identification depuis un lien externe), alors l'utilisateur sera redirigé à l'URL spécifiée dans loginRedirect.

Exemple :

```
<?php
    $this->Auth->loginRedirect = array('controller' => 'membres', 'action' =>
'accueil');
?>
```

5.2.6.6 logoutRedirect

Vous pouvez également spécifier où vous voulez que l'utilisateur soit redirigé après sa déconnexion, ayant pour action par défaut l'action de login.

```
<?php
    $this->Auth->logoutRedirect = array(Configure::read('Routing.admin') =>
false, 'controller' => 'membres', 'action' => 'logout');
?>
```

5.2.6.7 loginError

Change le message d'erreur par défaut affiché lorsque quelqu'un ne s'authentifie pas correctement.

```
<?php
    $this->Auth->loginError = "Non, vous vous êtes trompé! Ce n'est pas
le bon mot de passe!";
?>
```

5.2.6.8 authError

Change le message d'erreur par défaut affiché lorsque quelqu'un essaye d'accéder à une ressource ou une action qu'il n'est pas autorisé à accéder.

```
<?php
    $this->Auth->authError = "Désolé, vous n'avez pas les droits
suffisants.";
?>
```

5.2.6.9 autoRedirect

Normalement, le Composant Auth vous redirige automatiquement dès lors qu'il vous authentifie. Parfois, vous souhaitez faire d'autres vérifications avant de rediriger les utilisateurs :

```
<?php
    function beforeFilter() {
        ...
        $this->Auth->autoRedirect = false;
    }

    ...

    function login() {
    //-- le code de cette fonction ne s'exécute que lorsque autoRedirect est
défini à false (i.e. dans un beforeFilter).
        if ($this->Auth->user()) {
            if (!empty($this->data['Utilisateur']['se_souvenir_de_moi'])) {
                $cookie = array();
                $cookie['nom'] = $this->data['Utilisateur']['nom'];
                $cookie['motdepasse'] = $this->data['Utilisateur']['motdepasse'];
                $this->Cookie->write('Auth.Utilisateur', $cookie, true, '+2 weeks');
                unset($this->data['Utilisateur']['se_souvenir_de_moi']);
            }
            $this->redirect($this->Auth->redirect());
        }
```

```
        if (empty($this->data)) {
            $cookie = $this->Cookie->read('Auth.Utilisateur');
            if (!is_null($cookie)) {
                if ($this->Auth->login($cookie)) {
                    //  Efface le message auth, seulement si nous l'utilisons
                    $this->Session->delete('Message.auth');
                    $this->redirect($this->Auth->redirect());
                }
            }
        }
    }
?>
```

Le code de la fonction login ne s'exécutera pas *sauf si* vous définissez $autoRedirect à *false* dans un beforeFilter. Le code présent dans la fonction de login ne s'exécutera *qu'après* un essai d'authentification. C'est le meilleur endroit pour déterminer si oui ou non une connexion réussie a été effectuée par le Composant Auth (vous aurez peut-être envie d'enregistrer la dernière date d'authentification réussie, etc.).

5.2.6.10 authorize

Normalement, le Composant Auth essaiera de vérifier que les critères de login que vous avez saisis sont exacts, en les comparant à ce qui a été stocké dans votre modèle utilisateur. Cependant, vous voudrez peut-être certaines fois effectuer du traitement additionnel, en déterminant vos propres critères. En assignant à cette variable l'une des nombreuses valeurs possibles, vous pouvez faire différentes choses. En voici quelques-unes, parmi les plus communes, que vous souhaiterez peut-être utiliser.

```
<?php
    $this->Auth->authorize = 'controller';
?>
```

Lorsque authorize est défini à 'controller', vous aurez besoin d'ajouter une méthode appelée isAuthorized() à votre contrôleur. Cette méthode vous permet de faire plus de vérifications d'authentification et de retourner ensuite soit true, soit false.

```
<?php
    function isAuthorized() {
        if ($this->action == 'delete') {
            if ($this->Auth->user('role') == 'admin') {
                return true;
            } else {
                return false;
            }
        }

        return true;
    }
?>
```

Souvenez-vous que cette méthode sera inspectée, après que vous ayez déjà passé la vérification d'authentification simple du modèle utilisateur.

```
<?php
    $this->Auth->authorize = 'model';
```

```
?>
```

Vous ne souhaitez rien ajouter à votre contrôleur et peut-être utiliser les ACO's ? Vous pouvez demander au Composant Auth d'appeler une méthode, nommée isAuthorized(), dans votre modèle utilisateur, pour faire le même genre de choses :

```
<?php
    class Utilisateur extends AppModel {
        ...

        function isAuthorized($utilisateur, $controleur, $action) {

            switch ($action) {
                case 'default':
                    return false;
                    break;
                case 'delete':
                    if ($utilisateur['Utilisateur']['role'] == 'admin') {
                        return true;
                    }
                    break;
            }
        }
    }
?>
```

Enfin, vous pouvez utiliser authorize avec les actions, comme montré ci-dessous :

```
<?php
    $this->Auth->authorize = 'actions';
?>
```

En utilisant actions, Auth utilisera l'ACL et vérifiera avec AclComponent::check(). Une fonction isAuthorized n'est pas nésessaire.

```
<?php
    $this->Auth->authorize = 'crud';
?>
```

En utilisant crud, Auth utilisera l'ACL et vérifiera avec AclComponent::check(). Les actions devraient correspondre aux CRUD (voir mapActions).

5.2.6.11 sessionKey

Nom de la clé du tableau de session où l'enregistrement de l'utilisateur actuellement authentifié est stocké.

Par défaut, vaut "Auth", donc si non spécifié, l'enregistrement est stocké dans "Auth.{nom $modeleUtilisateur}".

```
<?php
    $this->Auth->sessionKey = 'Autorise';
?>
```

5.2.6.12 ajaxLogin

Si vous faites des requêtes Ajax ou Javascript qui nécessitent des sessions authentifiées, donnez à cette variable le nom d'un élément de vue que vous souhaiteriez rendre et retourner quand vous avez une session invalide ou expirée.

Comme dans toute partie de CakePHP, soyez certains d'avoir jeté une oeil à la classe AuthComponent dans l'API, pour avoir une vision plus approfondie du composant Auth.

5.2.6.13 authenticate

Si vous voulez utiliser un autre layout pour votre message d'erreur d'Authentification, vous pouvez le définir avec la variable flashElement. Cet autre élément sera utilisé pour l'affichage.

```
<?php
    $this->Auth->flashElement    = "message_erreur";
?>
```

5.2.7 allowedActions

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Set the default allowed actions to allow if setting the component to 'authorize' => 'controller'

```
var $components = array(
  'Auth' => array(
    'authorize' => 'controller',
    'allowedActions' => array('index','view','display');
  )
);
```

index, view, and display actions are now allowed by default.

5.3 Cookies

Il y a plusieurs variables de contrôleur qui vous permettent de configurer la façon dont les cookies sont crées et gérés. Définir ces variables spéciales dans la méthode beforeFilter() de votre contrôleur, vous permet de définir la façon dont fonctionne le Composant Cookie.

Valeur par défaut	Description	
string $name	'CakeCookie'	Le nom du cookie.
string $key	null	Cette chaine est utilisée pour encoder la valeur écrite dans le cookie. Elle doit être aléatoire et difficile à deviner.
string $domain	''	Le nom de domaine autorisé à accéder au cookie. Par ex : utilisez '.votredomaine.com' pour autoriser l'accès à tous vos sous-domaines.

int ou string $time	'5 Days'	Le temps d'expiration de votre cookie. Les nombres entiers sont interprétés comme des secondes et la valeur 0 est équivalente à 'cookie de session' : c'est à dire que le cookie expire quand le navigateur est fermé. Si une chaîne est définie, elle sera interprétée avec la fonction PHP strtotime(). Vous pouvez paramétrer cela directement dans la méthode write().
string $path	'/'	Le chemin sur le serveur où le cookie sera utilisé. Si $cookiePath est '/foo/', le cookie sera seulement utilisable dans le dossier /foo/ et dans tous ses sous-répertoires comme /foo/bar/ de votre domaine. La valeur par défaut est le domaine entier. Vous pouvez le paramétrer directement dans la méthode write().
boolean $secure	false	Indique que le cookie doit être transmis seulement par une connexion sécurisée HTTPS. Quand il vaut true, le cookie sera créé seulement si une connexion sécurisée existe. Vous pouvez le paramétrer directement dans la méthode write().

Le fragment de contrôleur suivant montre comment inclure le Composant Cookie et paramétrer les variables du contrôleur nécessaires pour écrire un cookie nommé 'boulanger_id', sur le domaine 'exemple.com' qui nécessite une connexion sécurisée, accessible via le chemin '/boulangers/preferences/' et qui expire dans une heure.

```
var $components    = array('Cookie');
function beforeFilter() {
  $this->Cookie->name = 'boulanger_id';
  $this->Cookie->time =  3600;  // ou '1 hour'
  $this->Cookie->path = '/boulangers/preferences/';
  $this->Cookie->domain = 'exemple.com';
  $this->Cookie->secure = true;  // envoyé seulement si connexion HTTPS
utilisée
  $this->Cookie->key = 'qSI232qs*&sXOw!';
}
```

Maintenant, voyons comment utiliser les différentes méthodes du Composant Cookie.

5.3.2 Utiliser le Composant

Cette section expose brièvement les méthodes du Composant Cookie.

write(mixed $key, mixed $value, boolean $encrypt, mixed $expires)

La méthode write() est le coeur du composant cookie, $key est le nom de la variable de cookie que vous souhaitez et $value est l'information à stocker.

```
$this->Cookie->write('nom','Larry');
```

Vous pouvez aussi regrouper vos variables en utilisant la notation avec point pour le paramètre key.

```
$this->Cookie->write('Utilisateur.nom', 'Larry');
 $this->Cookie->write('Utilisateur.role','Chef');
```

Si vous voulez écrire plus d'une valeur pour le cookie en une seule fois, vous pouvez passer un tableau :

```
$this->Cookie->write(
  array('nom'=>'Larry','role'=>'Chef')
  );
```

Toutes les valeurs du cookie sont cryptées par défaut. Si vous voulez stocker les valeurs en texte brut, définissez à false le troisième paramètre de la méthode write().

```
$this->Cookie->write('nom','Larry',false);
```

Le dernier paramètre pour write est $expires - nombre de secondes avant que votre cookie n'expire. Par commodité, ce paramètre peut aussi être passé sous forme d'une chaîne compréhensible par la fonction php strtotime() :

```
// Les deux cookies expirent dans une heure.
  $this->Cookie->write('prenom','Larry',false, 3600);
  $this->Cookie->write('nom','Masters',false, '1 hour');
```

read(mixed $key)

Cette méthode est utilisée pour lire la valeur d'une variable de cookie, dont le nom est spécifié par $key.

```
// Affiche "Larry"
  echo $this->Cookie->read('nom');

  // Vous pouvez aussi utiliser la notation avec point pour read
  echo $this->Cookie->read('Utilisateur.nom');

  // Pour récupérer sous forme de tableau, les variables que vous avez
groupées
  // en utilisant la notation avec point, faites quelque chose comme :
  $this->Cookie->read('Utilisateur');

  // ceci affiche quelque chose comme :
array('nom' => 'Larry', 'role'=>'Chef')
```

del(mixed $key)

Supprime une variable de cookie, dont le nom est $key. Fonctionne pour la notation avec point.

```
  // Supprime une variable
  $this->Cookie->del('bar')

  // Supprime la variable de cookie bar, mais pas tout ce qui se trouve sous foo
  $this->Cookie->del('foo.bar')
```

destroy()

Détruit le cookie courant.

5.4 Email

Ci-dessous les valeurs qu'on peut positionner avant d'appeler `EmailComponent::send()`

to
: adresse de destination (string)

cc
: tableau des adresses en copie du message

bcc
: tableau des adresses en copie cachée *bcc (blind carbon copy)*

replyTo
: adresse de réponse (string)

return
: adresse email de retour utilisée en cas d'erreur (string) (pour les erreurs de démon de mail : *mail-daemon/errors*)

from
: adresse de provenance (string)

subject
: sujet du message (string)

template
: l'élément email à utiliser pour le message (situé dans `app/views/elements/email/html/` et `app/views/elements/email/text/`)

layout
: le layout utilisé pour l'email (situé dans `app/views/layouts/email/html/` et `app/views/layouts/email/text/`)

lineLength
: longueur à laquelle les lignes doivent être coupées. Défaut à 70. (integer)

sendAs
: format auquel vous souhaitez envoyer le message (string, valeurs possibles : `text`, `html` ou `both`)

attachments
: tableau des fichiers à joindre (chemin relatif ou absolu)

delivery
: comment envoyer le message (`mail`, `smtp` [requiert le positionnement des smtpOptions ci-dessous] et `debug`)

smtpOptions
: tableau associatif d'options pour smtp mailer (`port`, `host`, `timeout`, `username`, `password`, `client`)

Il y a quelques autres choses qui peuvent être paramétrées, mais vous devriez vous référez à l'API pour plus d'informations

5.4.1.1 Envoyer des messages multiples dans une boucle

Si vous désirez envoyer des emails multiples dans une boucle, vous aurez besoin de re-initialiser les propriétés du composant avec la méthode reset. Cette initialisation doit précéder le positionnement des propriétés du nouvel email.

```
$this->Email->reset()
```

5.4.1.2 Debugging Emails

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

If you do **not** want to actually send an email and instead want to test out the functionality, you can use the following delivery option:

```
$this->Email->delivery = 'debug';
```

In order to view those debugging information you need to create an extra line in your view or layout file (e.g. underneath your normal flash message in /layouts/default.ctp):

```
<?php echo $this->Session->flash(); ?>
<?php echo $this->Session->flash('email'); ?>
```

5.4.2 Envoyer un message simple

Pour envoyer un message sans utiliser de template, passez simplement le corps du message comme une chaîne (ou un tableau de lignes) à la méthode send(). Par exemple :

```
$this->Email->from    = 'Quelqu\'un <quelqu-un@exemple.com>';
$this->Email->to      = 'Quelqu\'un d\'autre <quelqu-un-d-autre@exemple.com>';
$this->Email->subject = 'Test';
$this->Email->send('Corps du message !');
```

5.4.2.1 Mettre en place les mises en forme (layouts)

Pour obtenir à la fois des emails au format html et texte vous aurez besoin de créer deux mises en forme (layouts), comme lorsque vous mettez en place vos dispositions (layouts) pour l'affichage de vos vues dans le navigateur, vous devrez mettre en place des dispositions par défaut pour les messages email. Dans le répertoire `app/views/layouts/`, il vous faudra placer la structure minimum suivante :

```
    email/
        html/
            default.ctp
        text/
            default.ctp
```

Ce sont les fichiers qui contiennent les dispositions modèles (layout templates) par défaut pour vos messages. Des exemples ci-dessous :

`email/text/default.ctp`

```
    <?php echo $content_for_layout; ?>
```

`email/html/default.ctp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <body>
        <?php echo $content_for_layout; ?>
    </body>
</html>
```

5.4.2.2 Mettre en place un élément email pour le corps du message

Dans le répertoire de code `app/views/elements/email/` vous devrez créer deux répertoires pour `text` et `html` à moins que vous ne prévoyiez d'envoyer les messages que dans un des deux formats. Dans chacun de ces répertoires, vous devrez créer les patrons (templates) pour chaque type de message en se référant au contenu que vous envoyiez à la vue en utilisant soit $this->set() soit le paramètre $contents de la méthode send(). Quelques exemples simples sont montrés ci-dessous. Pour ces exemples on appelle le patron (template) simple_message.ctp

`text`

```
Cher <?php echo $User['first']. ' ' . $User['last'] ?>,
  Merci de votre intérêt.
```

`html`

```
<p>Cher <?php echo $User['first']. ' ' . $User['last'] ?>,<br />
   Merci de votre intérêt.</p>
```

5.4.2.3 Contrôleur

Dans votre contrôleur, vous devrez ajouter le composant au tableau de composants `$components` ou ajouter ce tableau $components à votre contrôleur de cette façon :

```
<?php
var $components = array('Email');
?>
```

Dans cet exemple, nous allons écrire une méthode privée pour prendre en charge les messages email vers un utilisateur identifié par son $id. Dans votre contrôleur (le contrôleur User dans cet exemple) :

```
<?php
function _envoiMailNouvelUtilisateur($id) {
    $Utilisateur = $this->Utilisateur ->read(null,$id);
    $this->Email->to = $Utilisateur ['Utilisateur']['email'];
    $this->Email->bcc = array('secret@exemple.com');
    $this->Email->subject = 'Bienvenue à ce truc très cool';
    $this->Email->replyTo = 'support@exemple.com';
    $this->Email->from = 'Appli Web Extra Cool <app@exemple.com>';
    $this->Email->template = 'simple_message'; // notez l'absence de '.ctp'
    // Envoi en 'html', 'text' ou 'both' (par défaut c'est 'text')
    $this->Email->sendAs = 'both'; // parce que nous aimons envoyer de jolis emails
    // Positionner les variables comme d'habitude
    $this->set('Utilisateur', $Utilisateur);
    // Ne passer aucun argument à send()
    $this->Email->send();
 }
?>
```

Voilà pour l'envoi du message. Vous pourriez appeler cette méthode depuis une autre méthode de cette façon :

```
$this->_envoiMailNouvelUtilisateur( $this->Utilisateur->id );
```

5.4.2.4 Pièce joint

Voici comment vous pouvez envoyer des pièces jointes avec votre message. Vous définissez un tableau contenant les chemins vers les fichiers à joindre à la propriété des pièces jointes de la composante.

```
$this->Email->attachments = array(
    TMP . 'foo.doc',
    'bar.doc' => TMP . 'Nom-du-fichier-réel'
);
```

Le premier fichier `foo.doc` sera joint avec le même nom. Pour le second fichier, nous indiquer un alias `bar.doc` sera être utilisé pour fixer la place de son nom réel du fichier `Nom-du-fichier-réel`

5.4.3 Envoyer un Message par SMTP

Pour envoyer un email en utilisant un serveur SMTP, les étapes sont similaires à l'envoi d'un message basique. Définissez la méthode de distribution à `smtp` et assignez toutes les options à la propriété `smtpOptions` de l'objet Email. Vous pouvez aussi récupérer les erreurs SMTP générées durant la session, en lisant la propriété `smtpError` du composant.

```
   /* Options SMTP */
   $this->Email->smtpOptions = array(
        'port'=>'25',
        'timeout'=>'30',
        'host' => 'votre.serveur.smtp',
        'username'=>'votre_login_smtp',
        'password'=>'votre_mot_de_passe_smtp',
        'client' => 'nom_machine_smtp_helo'
   );

    /* Définir la méthode de distribution */
    $this->Email->delivery = 'smtp';

    /* Ne passer aucun argument à send() */
    $this->Email->send();

    /* Vérification des erreurs SMTP. */
    $this->set('smtp-errors', $this->Email->smtpError);
```

Si votre serveur SMTP nécessite une authentification, assurez-vous de définir les parmètres nom d'utilisateur et mot de passe dans `smtpOptions`, comme indiqué dans l'exemple.

Si vous ne savez pas ce qu'est un HELO SMTP, alors vous ne devriez pas avoir besoin de définir le paramètre `client` dans `smtpOptions`. Celui-ci est seulement nécessaire pour les serveurs SMTP qui ne respectent pas pleinement la RFC 821 (SMTP HELO).

5.5 Gestion de requêtes

Le composant Request Handler est utilisé dans CakePHP pour obtenir des informations additionnelles au sujet des requêtes HTTP qui sont faites à votre application. Vous pouvez l'utiliser pour informer vos contrôleurs des process Ajax, tout autant que pour obtenir des informations complémentaires sur les types de contenus que le client accepte et modifier automatiquement le layout approprié, quand les extensions de fichier sont disponibles.

Par défaut, RequestHandler détectera automatiquement les requêtes Ajax basée 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 assistant avec le même nom que l'extension demandée, il sera ajouté au tableau d'assistant des Contrôleurs. Enfin, si une donnée XML est POST'ée vers vos Contrôleurs, elle sera décomposée en un objet XML, lequel sera assigné à Controller::data et pourra alors être sauvegardé comme une donnée de modèle. Afin d'utiliser le Request Handler il doit être inclus dans votre tableau $components.

```
<?php
class WidgetController extends AppController {

    var $components = array('RequestHandler');

    // suite du contrôleur
}
?>
```

5.5.1 Obtenir des informations sur une requête

Request Handler contient plusieurs méthodes qui nous donne des informations à propos du client et de ses requêtes.

accepts ($type = null)
$type peut être un string, un tableau , ou 'null'. Si c'est un string, 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 {

    var $components = array('RequestHandler');

    function beforeFilter () {
        if ($this->RequestHandler->accepts('html')) {
            // Execute le code seulement si le client accepte les réponse HTML
(text/html)
        } elseif ($this->RequestHandler->accepts('xml')) {
            // Execute seulement le code XML
        }
        if ($this->RequestHandler->accepts(array('xml', 'rss', 'atom'))) {
            // Execute seulement si le client accepte un des suivants: XML, RSS ou Atom
        }
    }
}
```

D'autres méthodes de détections du contenu des requêtes:

isAjax()
Renvoie true si la requête contient une réponse XMLHttpRequest.

isSSL()
Renvoie true si la requête a été faite à travers une connection SSL.

isXml()
Renvoie true si la requête actuelle accepte les réponses XML.

isRss()
Renvoie true si la requête actuelle accepte les réponses RSS.

isAtom()
Renvoie true si l'appel accepte les réponse Atom, false dans le cas contraire.

isMobile()
Renvoie true si le navigateur du client correspond à un téléphone portable, ou si sle client accepte le contenu WAP. Les navigateurs mobiles supportés sont les suivants:

- iPhone
- MIDP
- AvantGo
- BlackBerry
- J2ME
- Opera Mini
- DoCoMo
- NetFront
- Nokia
- PalmOS
- PalmSource
- portalmmm
- Plucker
- ReqwirelessWeb
- SonyEricsson
- Symbian
- UP.Browser
- Windows CE
- Xiino

isWap()
Returns true if the client accepts WAP content. Renvoie true si le client accepte le contenu WAP.

Toutes les méthodes de détection des requêtes précédentes peuvent être utilisée dans un contexte similaire pour filtrer les fonctionnalités destiné à du contenu spécifique.Par exemple, au moment de répondre aux requêtes AJAX, si vous voulez vider le cache du navigateur, et changer le niveau de débogage. Cependant, si vous voulez ne pas vider le cache pour les requêtes non-AJAX. , le code suivant vous permettra de le faire:

```
    if ($this->RequestHandler->isAjax()) {
        Configure::write('debug', 0);
        $this->header('Pragma: no-cache');
        $this->header('Cache-control: no-cache');
        $this->header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
    }
    //Continue Controller action
```

Vous pouvez aussi vider le cache grâce à cette fonction analogue `Controller::disableCache()`.

```
    if ($this->RequestHandler->isAjax()) {
        $this->disableCache();
    }
    //Action du contrôleur
```

5.5.2 Détection du type de requête

RequestHandler fournit aussi des informations quant au type des requêtes HTTP qui ont été faites et vous autorise à répondre à chaque type de requêtes.

`isPost()`
Renvoie true si la requête est de type POST.

`isPut()`
Renvoie true si la requête est de type PUT.

`isGet()`
Renvoie true si la requête est de type GET.

`isDelete()`
Renvoie true si la requête est de type DELETE.

5.5.3 Obtenir des informations supplémentaires sur le client

`getClientIP()`
Renvoie l'adresse IP du client.

`getReferer()`
Renvoie le nom de domaine à partir duquel la requête a été faite.

`getAjaxVersion()`
Renvoie 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' contient un header HTTP spécifique qui permet de déterminer sa version.

5.5.4 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.

`setContent($name, $type = null)`

- $name string - Le nom du type de contenu (*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. De plus, ces types de contenu sont utilisées par prefers() et accepts().

setContent est bien mieux utilisé dans le beforeFilter() de vos contrôleurs, 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

`prefers($type = null)`
Détermine quels content-types préfère le client. 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.

`renderAs($controller, $type)`

- $controller - Référence du contrôleur
- $type - nom simplifié du type de contenu à rendre, par exemple : xml, rss.

Change le mode de rendu d'un contrôleur pour le type spécifié. Ajoutera aussi l'assistant (*helper*) approprié au tableau des assistants du contrôleur, s'il est disponible et qu'il n'est pas déjà dans le tableau.

`respondAs($type, $options)`

- $type - nom simplifié du type de contenu à rendre, par exemple : xml, rss ou un content-type complet, tel application/x-shockwave
- $options - 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. Si DEBUG est plus grand que 2, l'en-tête n'est pas défini.

`responseType()`
Retourne l'en-tête Content-type du type de réponse courant ou null s'il y en a déjà un de défini.

`mapType($ctype)`
Rétro-associe un content-type à un alias

5.6 Composant Security

Le composant Security offre une manière simple d'inclure une sécurité renforcée à votre application. Une interface pour gérer les requêtes HTTP-authentifiées peut être créée avec le composant Security. Il est défini dans le beforeFilter() de vos contrôleurs. Il possède de nombreux paramètres configurables. Toutes ces propriétés peuvent être définies directement ou par l'intermédiaire de méthodes *setters* du même nom.

Si une action est restreinte par le composant 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 contrôleur. Gardez à l'esprit que ces trous noirs, issus de toutes les méthodes du composant Security, seraient exécutés à travers cette méthode de rappel.

Quand vous utilisez le composant Security, vous **devez** utiliser l'assistant Form pour créer vos formulaires. Le composant Security recherche certains indicateurs qui sont créés et gérés par l'assistant Form (spécialement ceux créés dans create() et end()).

5.6.1 Configuration

$blackHoleCallback
: Un contrôleur de callback qui gèrera les requêtes envoyées dans le "trou noir"

$requirePost
: Liste d'actions de contrôleurs qui exigent que se produise une requête POST. Un tableau d'actions de contrôleurs ou '*' pour forcer toutes les actions à exiger un POST.

$requireSecure
: Liste d'actions qui exigent que se produise une connexion SSL. Un tableau d'actions de contrôleurs ou '*' pour forcer toutes les actions à exiger une connexion SSL.

$requireAuth
: Liste d'actions qui exigent une clé d'authentification valide. Cette clé de validation est définie par le Composant Security.

$requireLogin
: Liste d'actions qui exigent des connexions HTTP Authentifiées (basic ou digest). Acceptent aussi '*' pour indiquer que toutes les actions de ce contrôleur exigent une authentification HTTP.

$loginOptions
: Les options pour les requêtes de connexion HTTP authentifiée. Vous permet de définir le type d'authentification et le contrôleur de callback pour le processus d'authentification.

$loginUsers
: Un tableau associatif de noms_utilisateurs => mots_de_passe qui est utilisé pour les connexions HTTP authentifiées. Si vous utilisez l'authentification digest, votre mot de passe devra être de hashage MD5

$allowedControllers
: Une liste de contrôleurs à partir desquelles les actions du contrôleur courant sont autorisées à recevoir des requêtes. Ceci peut être utilisé pour contrôler les demandes croisées de contrôleur.

$allowedActions
: Les actions parmi celles du contrôleur courant qui sont autorisées à recevoir des requêtes. Ceci peut être utilisé pour contrôler les demandes croisées de contrôleur.

$disabledFields
: Liste des champs de formulaire qui devraient être ignorés lors de la validation du POST. La valeur, présence ou absence de ces champs de formulaire, ne sera pas prise en compte lors de la vérification de la validité de soumission du formulaire. Spécifiez les champs comme vous le faites pour l'assistant Form (Model.nomduchamp).

5.6.2 Méthodes

Génère le texte pour l'en-tête d'une requête HTTP-Authentifiée, d'après un tableau $options.

$options contient généralement un 'type', un 'realm' (domaine). Le type indique quelle méthode HTTP-Authentifiée utiliser. Le domaine par défaut est lié à l'environnement actuel du serveur HTTP.

5.6.2.7 parseDigestAuthData(string $digest)

Utilisé le componsant Security est généralement fait dans la méthode beforeFilter() de votre contrôleur. Vous pouvez spécifier les restrictions de sécurité que vous voulez et le composant Security les forcera au démarrage

```
<?php
class WidgetController extends AppController {

    var $components = array('Security');

    function beforeFilter() {
        $this->Security->requirePost('delete');
    }
}
?>
```

Dans cette exemple, l'action delete peut être effectuée avec succès si celui ci reçoit une requête POST

```
<?php
class WidgetController extends AppController {

    var $components = array('Security');

    function beforeFilter() {
        if(isset($this->params[Configure::read('Routing.admin')])){
            $this->Security->requireSecure();
        }
    }
}
?>
```

Cette exemple forcera toutes les actions qui proviennent de la "route" Admin à être effectuées via des requêtes sécurisées SSL.

5.6.4 Authentification HTTP Basic

Le composant Security a quelques fonctions d'authentifications très puissantes. Parfois, vous pouvez avoir besoin de protéger quelques fonctionnalités de votre application, en utilisant une authentification HTTP Basic. L'un des usages courant de l'authentification HTTP est la protection d'une API REST ou SOAP.

Ce type d'authentification est appelée basic pour une raison simple. A moins que vous ne transfériez les informations par SSL, les identifiants seront transférés en texte brut.

Utiliser le composant Security pour les authentifications HTTP est facile. L'exemple suivant inclue le composant Security et quelques lignes de code dans la méthode beforeFilter() du contrôleur.

```
class ApiController extends AppController {
    var $name = 'Api';
    var $uses = array();
    var $components = array('Security');

    function beforeFilter() {
        $this->Security->loginOptions = array(
            'type'=>'basic',
            'realm'=>'MonDomaine'
        );
        $this->Security->loginUsers = array(
            'john'=>'mot_passe_john',
            'jane'=>'mot_passe_jane'
        );
        $this->Security->requireLogin();
    }

    function index() {
        // Logique protégée de l'application ici...
    }
}
```

La propriété loginOptions du composant Security est un tableau associatif qui spécifie comment le login devrait être manipulé. Vous avez uniquement besoin de spécifier le **type** comme **basic**, pour que cela fonctionne. Spécifier le **realm** (domaine) si vous voulez afficher un joli message à quiconque essaiera de s'identifier ou si vous avez plusieurs sections avec authentification dans votre application (= *realms*) que vous voulez garder séparées.

La propriété loginUsers du composant Security est un tableau associatif contenant les utilisateurs qui peuvent accéder à ce domaine et leurs mots de passe. Les exemples montrés ici utilisent des informations client codées en dur, mais vous voudrez certainement utiliser un modèle pour rendre vos identifiants d'authentification plus manageables.

Enfin, requireLogin() indique au composant Security que ce contrôleur nécessite une identification. Comme avec requirePost(), ci-dessus, fournir des noms aux méthodes protègera celles-ci tout en laissant les autres accessibles.

5.7 Sessions

Le composant 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 persister de différentes façons. Par défaut, elles utilisent les paramètres fournis par PHP, cependant, d'autres options existent.

cake
: Sauvegarde les fichiers de session dans votre dossier tmp/sessions de app.

database
: Utilise les sessions en base de données de CakePHP

cache
: Utilise le moteur de cache configuré par Cache::config(). Très utile quand utilisé conjointement avec Memcache (dans les configurations avec des multiples serveurs d'applications) pour stocker à la fois les données mises en cache et les sessions.

php
: C'est le paramètre par défaut. Sauvegarde les fichiers de session comme indiqué dans php.ini

Pour changer la méthode de manipulation des session par défaut, changez la configuration de Session.save selon vos objectifs. Si vous choisissez 'database', vous devriez aussi décommenter les paramètres de Session.datababase et exécuter le fichier SQL de base de données de session qui se trouve dans app/config.

5.7.1 Méthodes

Le composant 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. Par exemple, Utilisateur.identifiant se réfèrera au tableau suivant :

```
    array('Utilisateur' =>
            array('identifiant' => 'ClarkKent@dailyplanet.com')
    );
```

Les points sont utilisés pour indiquer les tableaux imbriqués. Cette notation est utilisée pour toutes les méthodes du composant Session dans lesquelles une variable $name est utilisée.

5.7.1.1 write

`write($name, $value)`

Ecrit dans la Session, en mettant $value dans $name. $name peut-être un tableau séparé par un point. Par exemple :

```
$this->Session->write('Personne.couleurYeux', 'Vert');
```

Cela écrit la valeur 'Vert' dans la session sous Personne => couleurYeux.

5.7.1.2 setFlash

`setFlash($message, $element = 'default', $params = array(), $key = 'flash')`

Utilisé pour définir une variable de session qui peut être utilisée pour un rendu dans la Vue. $element vous permet de contrôler quel élément (situé dans `/app/views/elements`) doit être utilisé pour le rendu du message. Si vous laissez `$element` sur 'default', le message sera encadré par ce qui suit :

```
<div id="flashMessage" class="message"> [message] </div>
```

$params vous permet de passer des variables additionnelles pour le gabarit rendu. $key place l'index $messages dans le tableau Message. Par défaut, c'est 'flash'.

Des paramètres peuvent être passés pour modifier le `div` rendu, par exemple, ajouter "class" dans le tableau $params, affectera une classe CSS au `div` de sortie, en utilisant `$session->flash()` dans votre gabarit ou votre vue.

```
$this->Session->setFlash('Message textuel d\'exemple', 'default', array('class'
=> 'classe_exemple'))
```

Le résultat après utilisation de `$session->flash()` avec l'exemple ci-dessus devrait être :

```
<div id="flashMessage" class="classe_exemple">Message textuel d'exemple</div>
```

5.7.1.3 read

`read($name)`

Retourne la valeur de $name dans la session. Si $name vaut null, la session entière sera retournée. Par ex :

```
$vert = $this->Session->read('Personne.couleurYeux');
```

Récupère la valeur "vert" dans la session.

5.7.1.4 check

`check($name)`

Utilisé pour vérifier qu'une variable de Session a été créée. Retourne vrai si la variable existe et faux dans le cas contraire.

5.7.1.5 delete

`delete($name) /*ou*/ del($name)`

Supprime les données de Session de $name. Par ex :

```
$this->Session->del('Personne.couleurYeux');
```

Notre donnée de session n'a plus la valeur 'Vert' ni même l'index couleurYeux attribué. Cependant, le modèle Personne est toujours dans la Session. Pour supprimer de la session toutes les informations de Personne, utilisez :

```
$this->Session->del('Personne');
```

5.7.1.6 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 ensuite en créer une nouvelle.

```
$this->Session->destroy()
```

5.7.1.7 error

`error()`

Utilisé pour déterminer la dernière erreur dans une session.

6 Comportements intégrés

Les Comportements (*Behaviors*) ajoutent des fonctionnalités supplémentaires à vos modèles. CakePHP est livré avec un certain nombre de comportements intégrés tels que : *Tree* et *Containable*.

6.1 ACL

Le comportement Acl fournit une solution pour intégrer sans souci un modèle dans votre système ACL. Il peut créer à la fois les AROs et les ACOs de manière transparente.

Pour utiliser le nouveau comportement, vous pouvez l'ajouter à la propriété $actsAs de votre modèle. 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 Utilisateur extends AppModel {
    var $actsAs = array('Acl' => array('type' => 'requester'));
}
```

Ceci attacherait le comportement Acl en mode ARO. Pour joindre le comportement ACL dans un mode ACO, utilisez :

```
class Post extends AppModel {
    var $actsAs = array('Acl' => array('type' => 'controlled'));
}
```

Vous pouvez aussi attacher le comportement à la volée, comme ceci :

```
    $this->Post->Behaviors->attach('Acl', array('type' => 'controlled'));
```

6.1.1 Utiliser le Comportement Acl

La plupart des tâches du comportement Acl sont réalisées de façon transparente, dans le callback afterSave() de votre modèle. Cependant, son utilisation nécessite que votre Modèle ait une méthode parentNode() définie. Ceci est utilisé par le comportement Acl, pour déterminer les relations parent->enfant. Une méthode parentNode() de modèle doit retourner null ou une référence au Modèle parent.

```
function parentNode() {
    return null;
}
```

Si vous voulez définir un noeud ACO ou ARO comme parent pour votre Modèle, parentNode() doit retourner l'alias du noeud ACO ou ARO.

```
function parentNode() {
        return 'noeud_racine';
}
```

Voici un exemple plus complet. Utilisons un modèle exemple Utilisateur, avec Utilisateur belongsTo Groupe.

```
function parentNode() {
    if (!$this->id && empty($this->data)) {
        return null;
    }
    $data = $this->data;
    if (empty($this->data)) {
        $data = $this->read();
    }
    if (!$data['Utilisateur']['groupe_id']) {
        return null;
    } else {
        $this->Groupe->id = $data['Utilisateur']['groupe_id'];
        $noeudGroupe = $this->Groupe->node();
        return array('Groupe' => array('id' => $noeudGroupe[0]['Aro']['foreign_key']));
    }
}
```

Dans l'exemple ci-dessus, le retour est un tableau qui ressemble aux résultats d'un find de modèle. Il est important d'avoir une valeur d'id définie ou bien la relation parentNode échouera. Le comportement Acl utilise ces données pour construire sa structure en arbre.

6.1.2 node()

Le Comportement Acl vous permet aussi de récupérer le noeud Acl associé à un enregistrement de modèle. Après avoir défini $model->id. Vous pouvez utiliser $model->node() pour récupérer le noeud Acl associé.

Vous pouvez aussi récupérer le noeud Acl de n'importe quelle ligne, en passant un tableau de données en paramètre.

```
    $this->Utilisateur->id = 1;
    $noeud = $this->Utilisateur->node();

    $utilisateur = array('Utilisateur' => array(
        'id' => 1
    ));
    $noeud = $this->Utilisateur->node($utilisateur);
```

Ces deux exemples retourneront la même information de noeud Acl.

6.2 Containable

Une nouvelle intégration au coeur de CakePHP est le Comportement "Containable". Ce comportement de modèle 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 class vous aidera aussi a chercher et filtrer vos données pour vos utilisateurs d'une façon propre et consistante.

Pour utiliser le nouveau comportement, vous pouvez l'ajouter à la propriété $actAs de votre modèle :

```
class Post extends AppModel {
    var $actsAs = array('Containable');
}
```

Vous pouvez aussi attacher le comportement à la volée :

```
$this->Post->Behaviors->attach('Containable');
```

Pour voir comment Containable fonctionne, regardons quelques exemples. Premièrement, nous commencerons avec un appel find() sur un modèle nommé Post. Disons que ce Post a plusieurs (hasMany) Comment, 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
                    [title] => First 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] => First comment
                            [created] => 2008-05-18 00:00:00
                        )
                    [1] => Array
                        (
                            [id] => 2
                            [post_id] => 1
                            [author] => Sam
                            [email] => sam@example.net
                            [website] => http://example.net
                            [comment] => Second comment
                            [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
                (...
```

For some interfaces in your application, you may not need that much information from the Post model. One thing the ContainableBehavior does is help you cut down on what find() returns.

For example, to get only the post-related information, you can do the following:

```
$this->Post->contain();
$this->Post->find('all');
```

You can also invoke Containable's magic from inside the find() call:

```
$this->Post->find('all', array('contain' => false));
```

Having done that, you end up with something a lot more concise:

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => First article
                    [content] => aaa
                    [created] => 2008-05-18 00:00:00
                )
        )
[1] => Array
        (
            [Post] => Array
                (
                    [id] => 2
                    [title] => Second article
                    [content] => bbb
                    [created] => 2008-05-19 00:00:00
                )
        )
```

This sort of help isn't new: in fact, you can do that without the ContainableBehavior doing something like this:

```
$this->Post->recursive = -1;
$this->Post->find('all');
```

Containable really shines when you have complex associations, and you want to pare down things that sit at the same level. The model's $recursive property is helpful if you want to hack off an entire level of recursion, but not when you want to pick and choose what to keep at each level. Let's see how it works by using the contain() method. The contain method's first argument accepts the name, or an array of names, of the models to keep in the find operation. If we wanted to fetch all posts and their related tags (without any comment information), we'd try something like this:

```
$this->Post->contain('Tag');
$this->Post->find('all');
```

Again, we can use the contain key inside a find() call:

```
$this->Post->find('all', array('contain' => 'Tag'));
```

Without Containable, you'd end up needing to use the unbindModel() method of the model, multiple times if you're paring off multiple models. Containable creates a cleaner way to accomplish this same task.

Containable also goes a step deeper: you can filter the data of the *associated* models. If you look at the results of the original find() call, notice the author field in the Comment model. If you are interested in the posts and the names of the comment authors—and nothing else—you could do something like the following:

```
$this->Post->contain('Comment.author');
$this->Post->find('all');

//or..

$this->Post->find('all', array('contain' => 'Comment.author'));
```

Here, we've told Containable to give us our post information, and just the author field of the associated Comment model. The output of the find call might look something like this:

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => First article
                    [content] => aaa
                    [created] => 2008-05-18 00:00:00
                )
            [Comment] => Array
                (
                    [0] => Array
                        (
                            [author] => Daniel
                            [post_id] => 1
                        )
                    [1] => Array
                        (
                            [author] => Sam
                            [post_id] => 1
                        )
                )
        )
[1] => Array
        (...
```

As you can see, the Comment arrays only contain the author field (plus the post_id which is needed by CakePHP to map the results).

You can also filter the associated Comment data by specifying a condition:

```
$this->Post->contain('Comment.author = "Daniel"');
$this->Post->find('all');

//or...

$this->Post->find('all', array('contain' => 'Comment.author = "Daniel"'));
```

This gives us a result that gives us posts with comments authored by Daniel:

```
[0] => Array
        (
            [Post] => Array
                (
                    [id] => 1
                    [title] => First 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] => First comment
                            [created] => 2008-05-18 00:00:00
                        )
                )
        )
```

Additional filtering can be performed by supplying the standard `Model->find()` options:

```
$this->Post->find('all', array('contain' => array(
    'Comment' => array(
        'conditions' => array('Comment.author =' => "Daniel"),
        'order' => 'Comment.created DESC'
    )
)));
```

Here's an example of using the Containble behavior when you've got deep and complex model relationships.

Let's consider the following model associations:

```
User->Profile
User->Account->AccountSummary
User->Post->PostAttachment->PostAttachmentHistory->HistoryNotes
User->Post->Tag
```

This is how we retrieve the above associations with 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%')
            )
        )
    )
));
```

Keep in mind that 'contain' key is only used once in the main model, you don't use 'contain' again for related models

When using 'fields' and 'contain' options - be careful to include all foreign keys that your query directly or indirectly requires. Please also note that because Containable must to be attached to all models used in containment, you may consider attaching it to your AppModel.

Here's an example of how to contain associations when paginating.

```
$this->paginate['User'] = array(
    'contain' => array('Profile', 'Account'),
    'order' => 'User.username'
);

$users = $this->paginate('User');
```

Utiliser Containable

Pour voir comment Containable fonctionne, regardons quelques exemples. D'abord, nous commencerons par un appel à find() sur un modèle nommé Post. Disons que Post hasMany Commentaire et Post hasAndBelongsToMany Tag. La quantité de données récupérées dans un appel à find() normal est plutôt vaste :

```
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@exemple.com
                        [site_web] => http://exemple.com
                        [commentaire] => Premier commentaire
                        [created] => 2008-05-18 00:00:00
                    )
                [1] => Array
                    (
                        [id] => 2
                        [post_id] => 1
                        [auteur] => Sam
                        [email] => sam@exemple.net
                        [site_web] => http://exemple.net
                        [commentaire] => Second commentaire
                        [created] => 2008-05-18 00:00:00
                    )
            )
        [Tag] => Array
            (
                [0] Array
                    (
                        [id] => 1
                        [nom] => Grandiose
                    )
                [1] => Array
                    (
                        [id] => 2
                        [nom] => Cuisson
                    )
            )
    [1] => Array
    (
        [Post] => Array
            (...
```

Pour certaines interfaces de votre application, vous n'aurez peut-être pas besoin d'autant d'informations issues du modèle Post. L'une des choses que réalise le `ContainableBehavior`, c'est de vous aider à réduire ce que retourne un find().

Par exemple, pour obtenir uniquement les informations liées à un post, vous pouvez faire la chose suivante :

```
$this->Post->contain();
$this->Post->find('all');
```

Vous pouvez aussi invoqué la magie de Containable à l'intérieur de l'appel à find() :

```
$this->Post->find('all', array('contain' => false));
```

En faisant çà, 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
                )
        )
```

Ce type d'optimisation n'est pas nouveau : en fait, vous pouvez réaliser çà sans le comportement `Containable`, en faisant quelque chose comme ceci :

```
$this->Post->recursive = -1;
$this->Post->find('all');
```

Containable se distingue réellement, lorsque vous avez des associations complexes et que vous voulez réduire les choses qui se situent au même niveau. La propriété de modèle `$recursive` est pratique si vous voulez déconnecter un niveau de récursion entier, mais pas lorsque vous voulez sélectionner et choisir que garder à chaque niveau. Voyons comment cela fonctionne en utilisant la méthode `contain()`.

Le premier argument de la méthode contain accepte le nom, ou un tableau de noms, des modèles à conserver dans l'opération find. Si nous avions voulu récupérer tous les posts et leurs tags liés (sans aucune information de commentaire, nous aurions essayé quelque chose comme çà :

```
$this->Post->contain('Tag');
$this->Post->find('all');
```

Là encore, nous pouvons utiliser la clé contain à l'intérieur de l'appel à find() :

```
$this->Post->find('all', array('contain' => 'Tag'));
```

Sans Containable, si vous avez plusieurs modèles, vous finiriez par avoir besoin d'utiliser la méthode `unbindModel()` du modèle de nombreuses fois. Le comportement Containable offre une manière plus propre d'accomplir cette même tâche.

Limiter des associations plus profondes

Le comportement Containable fonctionne également à un niveau plus profond : vous pouvez filtrer les données des modèles *associés*. Si vous regardez les résultats d'un appel au find() original, vous remarquez le champ auteur dans le modèle Commentaire. Si vous êtes intéressé par les posts et les noms des auteurs de commentaire - et rien d'autre - vous pourriez faire quelque chose comme çà :

```
$this->Post->contain('Commentaire.auteur');
$this->Post->find('all');

// ou...

$this->Post->find('all', array('contain' => 'Commentaire.auteur'));
```

Ici, nous avons dit au Containable de nous transmettre des informations sur notre post et seulement le champ auteur du modèle associé Commentaire. La sortie de l'appel à find devrait ressembler à quelque chose comme çà :

```
[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 Commentaires contiennent seulement le champ auteur (plus le post_id qui est utilisé par CakePHP pour relier les résultats).

Vous pouvez aussi filtrer les données du Commentaire associé en spécifiant une condition :

```
$this->Post->contain('Commentaire.auteur = "Daniel"');
$this->Post->find('all');

// ou...

$this->Post->find('all', array('contain' => 'Commentaire.auteur = "Daniel"'));
```

Ceci nous donne un résultat qui contient les posts avec des commentaires rédigés par Daniel :

```
[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@exemple.com
                            [site_web] => http://exemple.com
                            [commentaire] => Premier commentaire
                            [created] => 2008-05-18 00:00:00
                        )
                )
        )
```

Des filtrages aditionnels peuvent être effectués, en passant les options standards de `Model->find()` :

```
$this->Post->find('all', array('contain' => array(
    'Commentaire' => array(
        'conditions' => array('Commentaire.auteur =' => "Daniel"),
        'order'    => 'Commentaire.created DESC',
    )
)));
```

Voici un exemple d'utilisation du comportement `Containable`, lorsque vous avez des relations profondes et complexes entre modèles.

Considérons les associations de modèles suivantes :

```
Utilisateur->Profil
Utilisateur->Compte->SyntheseCompte
Utilisateur->Post->PieceJointePost->HistoriquePieceJointePost->NoteHistorique
Utilisateur->Post->Tag
```

Voici comment nous récupérons les associations ci-dessus avec Containable :

```
$this->Utilisateur->find('all', array(
    'contain'=>array(
        'Profil',
        'Compte' => array(
            'SyntheseCompte'
        ),
        'Post' => array(
            'PieceJointePost' => array(
                'fields' => array('id', 'nom'),
                'HistoriquePieceJointePost' => array(
                    'NoteHistorique' => array(
                        'fields' => array('id', 'note')
                    )
                )
            ),
            'Tag' => array(
                'conditions' => array('Tag.nom LIKE' => '%joyeux%')
```

```
            )
        )
    )
));
```

Gardez à l'esprit que la clé `contain` est utilisée une seule fois dans le modèle principal, vous n'avez pas à utiliser 'contain' de nouveau pour les modèles liés.

Quand vous utilisez les options 'fields' et 'contain', prenez soin d'inclure toutes les clés étrangères que votre requête nécessite, directement ou indirectement. Merci de noter également que, puisque le comportement Containable doit être attaché à tous les modèles utilisés par la limitation, vous devriez envisager de l'attacher à votre AppModel.

6.2.1 Utiliser Containable avec la pagination

Voici un exemple sur la manière de limiter les associations en paginant.

```
$this->paginate['Utilisateur'] = array(
    'contain' => array('Profil', 'Compte'),
    'order' => 'Utilisateur.pseudo'
);

$utilisateurs = $this->paginate('Utilisateur');
```

En incluant le paramètre 'contain' dans la propriété `$paginate`, elle sera appliquée à la fois au find('count') et au find('all') réalisés dans le modèle.

Les options du comportement Containable

Le `ContainableBehavior` a plusieurs options qui peuvent être définies quand le comportement est attaché à un modèle. Ces paramètres vous permettent d'affiner le comportement de Containable et de travailler plus facilement avec les autres comportements.

- **recursive** (boolean, optional), définir à true pour permettre au comportement Containable, de déterminer automatiquement le niveau de récursivité nécessaire pour récupérer les modèles spécifiés et de paramétrer la récursivité du modèle 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 ContainableBehavior à l'exécution, en ré-attachant le comportement comme vu au chapitre Utiliser les comportements

Le comportement Containable peut quelque fois causer des problèles avec d'autres comportements ou des requêtes qui utilisent des fonctions d'aggrégations et/ou des clauses GROUP BY. Si vous obtenez des erreurs SQL invalides à cause du mélange de champs aggrégés et non-aggrégés, essayer de désactiver le paramètre `autoFields`.

```
$this->Post->Behaviors->attach('Containable', array('autoFields' => false));
```

6.3 Translate

Le comportement 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 comportement, pour l'utiliser dans n'importe quel modèle.

Si vous utilisez le comportement 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.

6.3.1 Initialiser les tables 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 gabarit change dans les futures versions de CakePHP. En restant fidèle à la console, cela garantira que vous ayez le gabarit correct.

```
./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 base 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.

6.3.2 Attacher le Comportement Translate à vos Modèles

Ajoutez-le à votre modèle en utilisant la propriété `$actsAs` comme dans l'exemple suivant.

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate'
    );
}
?>
```

Ceci ne produira encore rien, parce qu'il faut un couple d'options avant que que cela ne commence à fonctionner. Vous devez définir, quels champs du modèle courant devront être détectés dans la table de traduction que nous avons créée à la première étape.

6.3.3 Définir les Champs

Vous pouvez définir les champs en étendant simplement la valeur `'Translate'` avec un autre tableau, comme çà :

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'champUn', 'champDeux', 'etc'
        )
    );
}
?>
```

Après avoir fait cela (par exemple, en précisant "nom" comme l'un des champs), vous avez déjà terminé la configuration de base. Super ! D'après notre exemple courant, le modèle devrait maintenant ressembler à quelque chose comme çà :

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'nom'
        )
    );
}
?>
```

6.3.4 Conclusion

A partir de maintenant, chaque mise à jour/création d'un enregistrement fera que le TranslateBehavior copiera la valeur de "nom" dans la table de traduction (par défaut : i18n), avec la locale courante. Une "locale" est un identifiant de langue.

La *locale courante* est la valeur actuelle de `Configure::read('Config.language')`. La valeur de *Config.language* est assignée dans la Classe L10n - à moins qu'elle ne soit déjà définie. Cependant, le Comportement Translate vous autorise à surcharger ceci à la volée, ce qui permet à l'utilisateur de votre page de créer de multiples versions sans avoir besoin de modifier ses préférences. Plus d'information sur ce point dans la prochaine section.

6.3.5 Récupérer tous les enregistrements de traduction pour un champ

Si vous voulez avoir tous les enregistrements de traduction attachés à l'enregistrement de modèle courant, vous étendez simplement le *tableau champ* dans votre paramétrage du comportement, comme montré ci-dessous. Vous êtes complètement libre de choisir le nommage.

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'nom' => 'nomTraduction'
        )
    );
}
?>
```

Avec ce paramétrage, le résultat de votre find() devrait ressembler à quelque chose comme çà :

```
Array
(
     [Post] => Array
         (
             [id] => 1
             [nom] => Exemple d\'entrée
             [corps] => lorem ipsum...
```

```
            [locale] => fr_fr
        )

    [nomTraduction] => Array
        (
            [0] Array
                (
                    [id] => 1
                    [locale] => en_us
                    [model] => Post
                    [foreign_key] => 1
                    [field] => nom
                    [content] => Example entry
                )

            [1] => Array
                (
                    [id] => 2
                    [locale] => fr_fr
                    [model] => Post
                    [foreign_key] => 1
                    [field] => nom
                    [content] => Exemple d'entrée
                )

        )
)
```

Note : L'enregistrement de modèle contient un champ *virtuel* appelée "locale". Il indique quelle locale est utilisée dans ce résultat.

6.3.5.1 Utiliser la méthode bindTranslation

Vous pouvez aussi récupérer toutes les traductions seulement quand vous en avez besoin, en utilisant la méthode bindTranslation

`bindTranslation($fields, $reset)`

`$fields` est 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 ('nom' => 'nomTraduction'));
$this->Post->find('all', array ('recursive'=>1)); // il est nécessaire
d'avoir au moins un recursive à 1 pour que ceci fonctionne
```

Avec ce paramétrage, le résultat de votre find() devrait ressembler à quelque chose comme çà :

```
Array
(
    [Post] => Array
        (
            [id] => 1
            [nom] => Exemple d'entrée
            [corps] => lorem ipsum...
            [locale] => fr_fr
        )

    [nomTraduction] => Array
```

```
        (
            [0] Array
                (
                    [id] => 1
                    [locale] => en_us
                    [model] => Post
                    [foreign_key] => 1
                    [field] => nom
                    [content] => Example entry
                )

            [1] => Array
                (
                    [id] => 2
                    [locale] => fr_fr
                    [model] => Post
                    [foreign_key] => 1
                    [field] => nom
                    [content] => Exemple d'entrée
                )

        )
)
```

6.3.6 Sauvegarder dans une autre langue

Vous pouvez forcer le modèle qui utilise le TranslateBehavior à sauvegarder dans une autre langue que celle détectée.

Pour dire à un modèle dans quelle langue le contenu devra être sauvé, changez simplement la valeur de la propriété `$locale` du modèle, avant que vous ne sauvegardiez les données dans la base. Vous pouvez faire çà dans votre contrôleur ou vous pouvez le définir directement dans le modèle.

Exemple A : dans votre contrôleur

```
<?php
class PostsController extends AppController {
    var $name = 'Posts';

    function add() {
        if ($this->data) {
            $this->Post->locale = 'de_de'; // nous allons sauvegarder la version allemande
            $this->Post->create();
            if ($this->Post->save($this->data)) {
                $this->redirect(array('action' => 'index'));
            }
        }
    }
}
?>
```

Exemple B : dans votre modèle

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
```

```
            'nom'
        )
    );

    // Option 1) définir simplement la propriété directement
    var $locale = 'fr_fr';

    // Option 2) créer une méthode simple
    function setLangue($locale) {
        $this->locale = $locale;
    }
}
?>
```

6.3.7 Multiple Translation Tables

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

If you expect a lot entries you probably wonder how to deal with a rapidly growing database table. There are two properties introduced by TranslateBehavior that allow to specify which "Model" to bind as the model containing the translations.

These are **$translateModel** and **$translateTable**.

Lets say we want to save our translations for all posts in the table "post_i18ns" instead of the default "i18n" table. To do so you need to setup your model like this:

```
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'name'
        )
    );

    // Use a different model (and table)
    var $translateModel = 'PostI18n';
}
?>
```

Important is that you have to pluralize the table. It is now a usual model and can be treated as such and thus comes with the conventions involved. The table schema itself must be identical with the one generated by the CakePHP console script. To make sure it fits one could just initialize a empty i18n table using the console and rename the table afterwards.

6.3.7.1 Créer le modèle de traduction

Pour que cela fonctionne vous devez créer le fichier de l'actuel modèle dans le dossier des modèles. La raison est qu'il n'y a pas de propriété pour définir le *displayField* directement dans le modèle utilisant ce comportement.

Assurez vous de changer le `$displayField` en `'field'`.

```php
<?php
class PostI18n extends AppModel {
    var $displayField = 'field'; // Important
}
// Nom du fichier: post_i18n.php
?>
```

C'est tout ce qu'il faut. Vous pouvez aussi ajouter toutes les propriétés des modèles comme $useTable. Mais pour une meilleure cohérence nous pouvons faire cela dans le modèle qui utilise ce modèle de traduction. C'est là que l'option `$translateTable` entre en jeu.

6.3.7.2 Modification d'une Table

Si vous voulez changer le nom de la table, il vous suffit simplement de définir $translateTable dans votre modèle, comme ceci :

```php
<?php
class Post extends AppModel {
    var $name = 'Post';
    var $actsAs = array(
        'Translate' => array(
            'name'
        )
    );

    // Utilise un Modèle différent
    var $translateModel = 'PostI18n';

    // Utilise une table différente pour translateModel
    var $translateTable = 'post_translations';
}
?>
```

A noter que **vous ne pouvez pas utiliser $translateTable seul**. Si vous n'avez pas l'intention d'utiliser un `$translateModel` personnalisé, alors laissez cette propriété inchangée. La raison est qu'elle casserait votre configuration et vous afficherait un message "Missing Table" pour le modèle I18n par défaut, lequel est créé à l'exécution.

6.4 Tree

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, sans avoir à connaître les détails de l'implémentation technique - à moins que ça ne vous intéresse ;).

6.4.1 Pré-requis

Pour utiliser le comportement 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.

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 réordonner l'arbre (et sans doute d'autres opérations) échouera.

6.4.2 Utilisation basique

Le comportement en arbre de données (Tree behavior) 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, 'Mes  Catégories'
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(2, 'Fun', 1, 2, 15);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(3, 'Sport', 2, 3, 8)
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(4, 'Surf', 3, 4, 5);
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(5, 'Tricot extrême',
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(6, 'Amis', 2, 9, 14)
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(7, 'Gérard', 6, 10,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(8, 'Gwendoline', 6,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(9, 'Travail', 1, 16,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(10, 'Rapports', 9, 1
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(11, 'Annuel', 10, 18
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(12, 'Statut', 10, 20
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(13, 'Voyages', 9, 23
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(14, 'National', 13,
INSERT INTO `categories` (`id`, `name`, `parent_id`, `lft`, `rght`) VALUES(15, 'International',
```

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 contrôleur :

```
<?php
class CategoriesController extends AppController {

        var $name = 'Categories';

        function index() {
                $this->data = $this->Category->generatetreelist(null, null, null,
```

```
'   ');
                debug ($this->data); die;
        }
}
?>
```

et une définition de modèle encore plus simple :

```
<?php
// app/models/category.php
class Category extends AppModel {
    var $name = 'Category';
    var $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 :

- Mes Catégories
  - ♦ Fun
    - ◊ Sport
      - · Surf
      - · Tricto extrême
    - ◊ Amis
      - · Gérard
      - · Gwendoline
  - ♦ Travail
    - ◊ Rapports
      - · Annuel
      - · Statut
    - ◊ Voyages
      - · National
      - · International

6.4.2.1 Adding data

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

In the previous section, we used existing data and checked that it looked hierarchal via the method `generatetreelist`. However, usually you would add your data in exactly the same way as you would for any model. For example:

```
// pseudo controller code
$data['Category']['parent_id'] =  3;
$data['Category']['name'] =  'Skating';
$this->Category->save($data);
```

When using the tree behavior its not necessary to do any more than set the parent_id, and the tree behavior will take care of the rest. If you don't set the parent_id, the tree behavior will add to the tree making your new addition a new top level entry:

```
// pseudo controller code
$data = array();
$data['Category']['name'] =  'Other People\'s Categories';
$this->Category->save($data);
```

Running the above two code snippets would alter your tree as follows:

- My Categories
    - ♦ Fun
        - ◊ Sport
            - · Surfing
            - · Extreme knitting
            - · Skating **New**
        - ◊ Friends
            - · Gerald
            - · Gwendolyn
    - ♦ Work
        - ◊ Reports
            - · Annual
            - · Status
        - ◊ Trips
            - · National
            - · International
- Other People's Categories **New**

6.4.2.2 Modifying data

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Modifying data is as transparent as adding new data. If you modify something, but do not change the parent_id field - the structure of your data will remain unchanged. For example:

```
// pseudo controller code
$this->Category->id = 5; // id of Extreme knitting
$this->Category->save(array('name' =>'Extreme fishing'));
```

The above code did not affect the parent_id field - even if the parent_id is included in the data that is passed to save if the value doesn't change, neither does the data structure. Therefore the tree of data would now look like:

- My Categories
    - ♦ Fun
        - ◊ Sport
            - · Surfing
            - · Extreme fishing **Updated**
            - · Skating
        - ◊ Friends
            - · Gerald
            - · Gwendolyn
    - ♦ Work
        - ◊ Reports

            · Annual
            · Status
        ◊ Trips
            · National
            · International
- Other People's Categories

Moving data around in your tree is also a simple affair. Let's say that Extreme fishing does not belong under Sport, but instead should be located under Other People's Categories. With the following code:

```
// pseudo controller code
$this->Category->id = 5; // id of Extreme fishing
$newParentId = $this->Category->field('id', array('name' => 'Other People\'s
Categories'));
$this->Category->save(array('parent_id' => $newParentId));
```

As would be expected the structure would be modified to:

- My Categories
    - ♦ Fun
        - ◊ Sport
            - · Surfing
            - · Skating
        - ◊ Friends
            - · Gerald
            - · Gwendolyn
    - ♦ Work
        - ◊ Reports
            - · Annual
            - · Status
        - ◊ Trips
            - · National
            - · International
- Other People's Categories
    - ♦ Extreme fishing **Moved**

6.4.2.3 Deleting data

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The tree behavior provides a number of ways to manage deleting data. To start with the simplest example; let's say that the reports category is no longer useful. To remove it *and any children it may have* just call delete as you would for any model. For example with the following code:

```
// pseudo controller code
$this->Category->id = 10;
$this->Category->delete();
```

The category tree would be modified as follows:

- My Categories
    - ♦ Fun

- ◊ Sport
  - · Surfing
  - · Skating
- ◊ Friends
  - · Gerald
  - · Gwendolyn

♦ Work

- ◊ Trips
  - · National
  - · International

• Other People's Categories

♦ Extreme fishing

6.4.2.4 Interroger et utiliser vos données

Utiliser et manipuler des données hiérarchisées peut s'avérer assez difficile. C'est pourquoi le comportement tree met à votre disposition quelques méthodes de permutations en plus des méthodes find de bases.

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 faites une demande de tri sur un tree, vous risquez d'obtenir des résultats inattendus.

6.4.2.4.1 Children

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The `children` method takes the primary key value (the id) of a row and returns the children, by default in the order they appear in the tree. The second optional parameter defines whether or not only direct children should be returned. Using the example data from the previous section:

```
$allChildren = $this->Category->children(1); // a flat array with 11 items
// -- or --
$this->Category->id = 1;
$allChildren = $this->Category->children(); // a flat array with 11 items

// Only return direct children
$directChildren = $this->Category->children(1, true); // a flat array with 2 items
```

If you want a recursive array use `find('threaded')`

Parameters for this function include:

- **$id**: The ID of the record to look up
- **$direct**: Set to true to return only the direct descendants
- **$fields**: Single string field name or array of fields to include in the return
- **$order**: SQL string of ORDER BY conditions
- **$limit**: SQL LIMIT statement
- **$page**: for accessing paged results
- **$recursive**: Number of levels deep for recursive associated Models

6.4.2.4.2 childCount

Cette méthode compte les enfants d'un enregistrement donnée. Vous pouvez avoir le nombre total de descendants ou seulement les descendants directs.

```
// Compte tous les sous catégories descendantes de la catégorie avec l'id =
5
$numChildren = $this->Category->childCount(5);
// -- or --
$this->Category->id = 5;
$numChildren = $this->Category->childCount();

// Compte seulement les descendants directs
$numChildren = $this->Category->childCount(5, true);
```

6.4.2.4.3 generatetreelist

generatetreelist (&$model, $conditions=null, $keyPath=null, $valuePath=null, $spacer= '_', $recursive=null)

Cette méthode retourne des données similaires à un find('list'), avec un préfixe d'indentation pour mettre en évidence la structure de l'arbre. Voici un exemple de rendu de cette méthode.

```
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"
)
```

6.4.2.4.4 getparentnode

L'utilitée de cette fonction est, comme son nom l'indique, de retourner la node parente d'une node, ou *false* si la node n'as pas de node parente (node racine). Exemple:

```
$parent = $this->Category->getparentnode(2); //<- id pour fun
// $parent contient All categories
```

6.4.2.4.5 getpath

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

getpath($id = null, $fields = null, $recursive = null)

The 'path' when refering to hierachial data is how you get from where you are to the top. So for example the path from the category "International" is:

- My Categories
  - ♦ ...
  - ♦ Work
    - ◊ Trips
      - · ...
      - · International

Using the id of "International" getpath will return each of the parents in turn (starting from the top).

```
$parents = $this->Category->getpath(15);
```

```
// contents of $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', ..)),
)
```

6.4.3 Autres méthodes

Le comportement en arbre de données (Tree behavior) ne travaille pas seulement en arrière plan, il y a une certains nombres de méthodes spécifiques définies dans ce comportemant (bahavior) qui peuvent être appélées directement : Ci-dessous une description brève et un exemple pour chacune d'entre elles:

6.4.3.1 moveDown

Utilisé pour descendre un noeud dans l'arbre hiérarchique. Vous devez spécifier l'id de l'élément à descendre et un entier positif spécifiant de combien de positions le noeud devrait être descendu. Tous les sous-noeuds seront également déplacés dans l'arbre.

Ci-dessus un exemple d'une action d'un contrôleur (dans un contrôleur nommé Categories) qui déplace un noeud spécifique dans l'arbre hiérarchique:

```
function movedown($title = null, $delta = null) {
        $cat = $this->Category->findByTitle($title);
        if (empty($cat)) {
            $this->Session->setFlash('Aucune catégorie ne porte le nom ' .
$title);
            $this->redirect(array('action' => 'index'), null, true);
        }

        $this->Category->id = $cat['Category']['id'];

        if ($delta > 0) {
            $this->Category->moveDown($this->Category->id, abs($delta));
        } else {
            $this->Session->setFlash('Merci de préciser de combien de crans le
noeud doit être déplacé');
        }

        $this->redirect(array('action' => 'show'), null, true);
    }
```

Par exemple, si vous vouliez déplacer la catégories "Cookies" d'un cran vers la bas, votre requête serait : /categories/movedown/Cookies/1.

6.4.3.2 moveUp

Utilisé pour déplacer vers le haut un seul noeud dans l'arbre hiérarchique. Vous devez spécifier l'id de l'élément à déplacer et un entier positif spécifiant de combien de positions le noeud devra être déplacé. Tous les noeuds enfants seront également déplacés.

Ci-dessous un exemple d'une action de contrôleur (dans un contrôleur nommé Categories) qui déplace un noeud spécifique dans l'arbre :

```
function moveup($name = null, $delta = null){
        $cat = $this->Category->findByName($name);
        if (empty($cat)) {
            $this->Session->setFlash('Il n\'y a pas de catégorie
nommée ' . $name);
            $this->redirect(array('action' => 'index'), null, true);
        }

        $this->Category->id = $cat['Category']['id'];

        if ($delta > 0) {
            $this->Category->moveup($this->Category->id, abs($delta));
        } else {
            $this->Session->setFlash('Merci de préciser de combien de
positions la catégorie doit être montée.');
        }

        $this->redirect(array('action' => 'index'), null, true);

    }
```

Par exemple, si vous voulez déplacer la catégorie "Gwendolyn" d'un cran vers le haut, votre requête sera : /categories/moveup/Gwendolyn/1. Maintenant l'ordre de Friends sera Gwendolyn, Gerald.

6.4.3.3 removeFromTree

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`removeFromTree($id=null, $delete=false)`

Using this method wil either delete or move a node but retain its sub-tree, which will be reparented one level higher. It offers more control than `delete()`, which for a model using the tree behavior will remove the specified node and all of its children.

Taking the following tree as a starting point:

- My Categories
  - Fun
    - Sport
      - Surfing
      - Extreme knitting

· Skating

Running the following code with the id for 'Sport'

```
$this->Node->removeFromTree($id);
```

The Sport node will be become a top level node:

- My Categories
  - ♦ Fun
    - ◊ Surfing
    - ◊ Extreme knitting
    - ◊ Skating
- Sport **Moved**

This demonstrates the default behavior of `removeFromTree` of moving the node to have no parent, and re-parenting all children.

If however the following code snippet was used with the id for 'Sport'

```
$this->Node->removeFromTree($id,true);
```

The tree would become

- My Categories
  - ♦ Fun
    - ◊ Surfing
    - ◊ Extreme knitting
    - ◊ Skating

This demonstrates the alternate use for `removeFromTree`, the children have been reparented and 'Sport' has been deleted.

6.4.3.4 reorder

Cette méthode peut être utilisée pour trier hiérarchiquement les données.

6.4.4 Data Integrity

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

```
recover(&$model, $mode = 'parent', $missingParentAction = null)
```

The `mode` parameter is used to specify the source of info that is valid/correct. The opposite source of data will be populated based upon that source of info. E.g. if the MPTT fields are corrupt or empty, with the `$mode 'parent'` the values of the `parent_id` field will be used to populate the left and right fields. The `missingParentAction` parameter only applies to "parent" mode and determines what to do if the parent field contains an id that is not present.

Available `$mode` options:

Available `missingParentActions` options when using `mode='parent'`:

```
// Rebuild all the left and right fields based on the parent_id
$this->Category->recover();
// or
$this->Category->recover('parent');

// Rebuild all the parent_id's based on the lft and rght fields
$this->Category->recover('tree');
```

6.4.4.2 Reorder

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`reorder(&$model, $options = array())`

Reorders the nodes (and child nodes) of the tree according to the field and direction specified in the parameters. This method does not change the parent of any node.

Reordering affects all nodes in the tree by default, however the following options can affect the process:

`$options` is used to pass all extra parameters, and has the following possible keys by default, all of which are optional:

```
array(
    'id' => null,
    'field' => $model->displayField,
    'order' => 'ASC',
    'verify' => true
)
```

6.4.4.3 Verify

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`verify(&$model)`

Returns `true` if the tree is valid otherwise an array of errors, with fields for type, incorrect index and message.

Each record in the output array is an array of the form (type, id, message)

```
    $this->Categories->verify();
```

Example output:

```
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
        )

)
```

7 Assistants intégrés

Les Assistants (*Helpers*) sont des classes comme les composants pour la couche de présentation de votre application. Ils contiennent de la logique de présentation qui est partagée entre plusieurs vues, éléments ou *layouts*. Cette section décrit chacun de ces assistants intégrés à CakePHP tels que Form, Html, JavaScript et RSS. Lisez la rubrique Assistants pour en apprendre davantage sur eux et sur la manière dont vous pouvez créer vos propres assistants.

7.1 AJAX

L'assistant AJAX utilise les librairies populaires que sont Prototype et script.aculo.us pour les requêtes AJAX et les effets de slide côté client. Pour utiliser l'assistant AJAX, vous devez avoir la version actuelle de la librairie Javascript www.prototypejs.org et http://script.aculo.us placé dans /app/webroot/js/.De plus, vous devrez inclure les librairies Javascript Prototype et script.aculo.us dans chaque vues utilisant les fonctionnalités de l'assistant AJAX.

Vous devez inclure les assistants AJAX et Javascript dans votre contrôleur.

```
class WidgetsController extends AppController {
    var $name = 'Widgets';
    var $helpers = array('Html','Ajax','Javascript');
}
```

Une fois l'assistant Javascript inclus dans votre contrôleur, vous pouvez utiliser la méthode link() de l'assistant Javascript pour inclure Prototype et Scriptaculous:

```
echo $javascript->link('prototype');
echo $javascript->link('scriptaculous');
```

Vous pouvez désormais utiliser l'assistant AJAX dans votre vue:

```
$ajax->whatever();
```

Si le Composant RequestHandler est inclus dans le contrôleur, cakePHP appliquera automatiquement la mise en page AJAX quand une requête sera demandée.

```
class WidgetsController extends AppController {
    var $name = 'Widgets';
    var $helpers = array('Html','Ajax','Javascript');
    var $components = array( 'RequestHandler' );
}
```

7.1.1 Options de l'assistant AJAX

`$option` keys	Description
`$options['evalScripts']`	Détermine si les balises script dans le contenu retourné sont évaluées. Défini à *true* par défaut.

`$options['frequency']`	Le nombre de seconde entre les vérifications à intervalle régulier.
`$options['indicator']`	L'id DOM d'un élément à montrer lorsqu'une requête est en chargement et à cacher quand la requête est finie.
`$options['position']`	Pour insérer plutôt que remplacer, utilisez cette option pour préciser la position d'insertion entre *top*, *bottom*, *after* ou *before*.
`$options['update']`	L'id de l'élement du DOM qui sera mis à jour avec le contenu retourné.
`$options['url']`	L'url au format contrôleur/action que vous souhaitez appeler.
`$options['type']`	Indique si la requête doit être 'synchronous' (synchrone) ou 'asynchronous' (asynchrone, valeur par défaut).
`$options['with']`	Une chaîne "URL-encodée" qui sera ajoutée à l'URL pour les méthodes get ou dans le corps du post pour tout autre méthode. Exemple : `x=1&toto=tata&y=2`. Les paramètres seront disponibles dans `$this->params['form']` ou dans `$this->data`, en fonction du format. Pour plus d'information, voyez la méthode Serialize de Prototype.

7.1.1.2 Options Callback

Les options *Callbacks* vous permettent d'appeler des fonctions JavaScript à des endroits spécifiques. Si vous cherchez un moyen d'injecter un peu de logique, avant, après ou lors de vos opérations utilisant l'assistant Ajax, utilisez ces *callbacks* pour mettre les choses en place.

$options	Description
$options['condition']	Extrait de code JavaScript qui doit évaluer *true* avant que la demande soit initialisée.
$options['before']	Exécuté avant que la demande soit faite. Une utilisation courante de ce *callBack* est de permettre l'affichage d'un indicateur de progression.
$options['confirm']	Texte à afficher dans une alerte de confirmation JavaScript avant de continuer.
$options['loading']	*Callback* exécuté lorsque des données sont récupérées à partir du serveur.
$options['after']	JavaScript appelé après que la requête soit lancée et avant que le *callback* $options['loading'] se lance.
$options['loaded']	*Callback* exécuté lorsque le document distant a été reçu par le client.
$options['interactive']	Appelée lorsque l'utilisateur peut interagir avec le document distant, même si il n'a pas fini de se charger.
$options['complete']	*Callback* JavaScript à exécuter lorsque XMLHttpRequest est terminé.

7.1.2 Méthodes

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`link(string $title, mixed $href, array $options, string $confirm, boolean $escapeTitle)`

Returns a link to a remote action defined by `$options['url']` or `$href` that's called in the background using XMLHttpRequest when the link is clicked. The result of that request can then be inserted into a DOM object whose id can be specified with `$options['update']`.

If `$options['url']` is blank the href is used instead

Example:

```
<div id="post">
</div>
<?php echo $ajax->link(
    'View Post',
    array( 'controller' => 'posts', 'action' => 'view', 1 ),
    array( 'update' => 'post' )
);
?>
```

By default, these remote requests are processed asynchronously during which various callbacks can be triggered

Example:

```
<div id="post">
</div>
<?php echo $ajax->link(
    'View Post',
    array( 'controller' => 'posts', 'action' => 'post', 1 ),
    array( 'update' => 'post', 'complete' => 'alert( "Hello World" )'  )
);
?>
```

To use synchronous processing specify `$options['type'] = 'synchronous'`.

To automatically set the ajax layout include the *RequestHandler* component in your controller

By default the contents of the target element are replaced. To change this behaviour set the `$options['position']`

Example:

```
<div id="post">
</div>
<?php echo $ajax->link(
    'View Post',
    array( 'controller' => 'posts', 'action' => 'view', 1),
    array( 'update' => 'post', 'position' => 'top'  )
);
?>
```

`$confirm` can be used to call up a JavaScript confirm() message before the request is run. Allowing the user to prevent execution.

Example:

```
<div id="post">
</div>
<?php echo $ajax->link(
    'Delete Post',
    array( 'controller' => 'posts', 'action' => 'delete', 1 ),
    array( 'update' => 'post' ),
    'Do you want to delete this post?'
);
?>
```

7.1.2.2 remoteFunction

`remoteFunction(array $options);`

Cette fonction crée le code JavaScript nécessaire pour effectuer un appel distant. Elle est principalement utilisée en tant qu'assistant pour link(). Ceci n'est pas utilisé très souvent, à moins que vous n'ayez besoin de générer des scripts personnalisés.

Les `$options` pour cette fonction sont les mêmes que pour la méthode `link`

Exemple :

```
<div id="post">
</div>
<script type="text/javascript">
<?php echo $ajax->remoteFunction(
    array(
        'url' => array( 'controller' => 'posts', 'action' => 'voir', 1 ),
        'update' => 'post'
    )
); ?>
</script>
```

Il peut aussi être assigné à un attribut d'évènement HTML :

```
<?php
    $remoteFunction = $ajax->remoteFunction(
        array(
        'url' => array( 'controller' => 'posts', 'action' => 'voir', 1 ),
        'update' => 'post' )
    );
?>
<div id="post" onmouseover="<?php echo $remoteFunction; ?>" >
Bougez la souris ici
</div>
```

Si `$options['update']` n'est pas transmis, le navigateur ignorera la réponse du serveur.

7.1.2.3 remoteTimer

`remoteTimer(array $options)`

Appelle périodiquement l'action `$options['url']`, toutes les `$options['frequency']` secondes. Généralement utilisé pour mettre à jour un div spécifique (défini dans `$options['update']`) avec le résultat de l'appel distant. Les *Callbacks* peuvent être utilisés.

`remoteTimer` est identique à `remoteFunction` à l'exception du paramètre supplémentaire `$options['frequency']`

Exemple :

```
<div id="post">
</div>
<?php
echo $ajax->remoteTimer(
    array(
    'url' => array( 'controller' => 'posts', 'action' => 'voir', 1 ),
    'update' => 'post', 'complete' => 'alert( "requête terminée" )',
    'position' => 'bottom', 'frequency' => 5
    )
);
?>
```

La valeur par défaut de `$options['frequency']` est 10 secondes

7.1.2.4 form

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`form(string $action, string $type, array $options)`

Returns a form tag that submits to $action using XMLHttpRequest instead of a normal HTTP request via $type ('post' or 'get'). Otherwise, form submission will behave exactly like normal: data submitted is available at $this->data inside your controllers. If $options['update'] is specified, it will be updated with the resulting document. Callbacks can be used.

The options array should include the model name e.g.

```
$ajax->form('edit','post',array('model'=>'User','update'=>'UserInfoDiv'));
```

Alternatively, if you need to cross post to another controller from your form:

```
$ajax->form(array('type' => 'post',
    'options' => array(
        'model'=>'User',
        'update'=>'UserInfoDiv',
        'url' => array(
            'controller' => 'comments',
```

```
            'action' => 'edit'
        )
    )
));
```

You should not use the `$ajax->form()` and `$ajax->submit()` in the same form. If you want the form validation to work properly use the `$ajax->submit()` method as shown below.

7.1.2.5 submit

`submit(string $titre, array $options)`

Retourne un bouton *submit* qui soumet le formulaire à `$options['url']` et met à jour le div spécifié dans `$options['update']`

```
<div id='testdiv'>
<?php
echo $form->create('Utilisateurs');
echo $form->input('email');
echo $form->input('nom');
echo $ajax->submit('Soumettre', array('url'=> array('controller'=>'utilisateurs',
'action'=>'ajouter'), 'update' => 'testdiv'));
echo $form->end();
?>
</div>
```

Utilisez la méthode `$ajax->submit()` si vous voulez que la validation du formulaire fonctionne bien. A savoir : vous voulez que les messages indiqués dans vos règles de validation s'affiche correctement.

7.1.2.6 observeField

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`observeField(string $fieldId, array $options)`

Observes the field with the DOM id specified by $field_id (every $options['frequency'] seconds) and makes an XMLHttpRequest when its contents have changed.

```
<?php echo $form->create( 'Post' ); ?>
<?php $titles = array( 1 => 'Tom', 2 => 'Dick', 3 => 'Harry' ); ?>
<?php echo $form->input( 'title', array( 'options' => $titles ) ) ?>
</form>

<?php
echo $ajax->observeField( 'PostTitle',
    array(
        'url' => array( 'action' => 'edit' ),
        'frequency' => 0.2,
    )
);
?>
```

`observeField` uses the same options as `link`

The field to send up can be set using `$options['with']`. This defaults to `Form.Element.serialize('$fieldId')`. Data submitted is available at `$this->data` inside your controllers. Callbacks can be used with this function.

To send up the entire form when the field changes use `$options['with'] = Form.serialize( $('Form ID') )`

7.1.2.7 observeForm

`observeForm(string $form_id, array $options)`

Similaire à observeField(), mais fonctionne sur un formulaire complet, identifié par son id DOM $form_id. Les $options fournies sont les mêmes que observeField(), à l'exception de la valeur par défaut de l'option $options['with'], qui est évaluée à la valeur sérialisée (chaine de requête) du formulaire.

7.1.2.8 autoComplete

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`autoComplete(string $fieldId, string $url, array $options)`

Renders a text field with $fieldId with autocomplete. The remote action at $url should return a suitable list of autocomplete terms. Often an unordered list is used for this. First, you need to set up a controller action that fetches and organizes the data you'll need for your list, based on user input:

```
function autoComplete() {
    //Partial strings will come from the autocomplete field as
    //$this->data['Post']['subject']
    $this->set('posts', $this->Post->find('all', array(
                'conditions' => array(
                    'Post.subject LIKE' => $this->data['Post']['subject'].'%'
                ),
                'fields' => array('subject')
    )));
    $this->layout = 'ajax';
}
```

Next, create `app/views/posts/auto_complete.ctp` that uses that data and creates an unordered list in (X)HTML:

```
<ul>
 <?php foreach($posts as $post): ?>
     <li><?php echo $post['Post']['subject']; ?></li>
 <?php endforeach; ?>
</ul>
```

Finally, utilize autoComplete() in a view to create your auto-completing form field:

```
<?php echo $form->create('User', array('url' => '/users/index')); ?>
    <?php echo $ajax->autoComplete('Post.subject', '/posts/autoComplete')?>
<?php echo $form->end('View Post')?>
```

Once you've got the autoComplete() call working correctly, use CSS to style the auto-complete suggestion box. You might end up using something similar to the following:

```
div.auto_complete    {
     position         :absolute;
     width            :250px;
     background-color :white;
     border           :1px solid #888;
     margin           :0px;
     padding          :0px;
}
li.selected    { background-color: #ffb; }
```

7.1.2.9 isAjax

`isAjax()`

Vous permet de vérifier si la requête actuelle est une requête Ajax de Prototype à l'intérieur d'une vue. Renvoie un booléen. Peut être utilisé pour une logique de présentation, pour afficher/cacher des blocs de contenu.

7.1.2.10 drag & drop

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`drag(string $id, array $options)`

Makes a Draggable element out of the DOM element specified by $id. For more information on the parameters accepted in $options see http://github.com/madrobby/scriptaculous/wikis/draggable.

Common options might include:

$options keys	Description
$options['handle']	Sets whether the element should only be draggable by an embedded handle. The value must be an element reference or element id or a string referencing a CSS class value. The first child/grandchild/etc. element found within the element that has this CSS class value will be used as the handle.
$options['revert']	If set to true, the element returns to its original position when the drags ends. Revert can also be an arbitrary function reference, called when the drag ends.
$options['constraint']	Constrains the drag to either 'horizontal' or 'vertical', leave blank for no constraints.

`drop(string $id, array $options)`

Makes the DOM element specified by $id able to accept dropped elements. Additional parameters can be specified with $options. For more information see http://github.com/madrobby/scriptaculous/wikis/droppables.

Common options might include:

$options keys	Description
$options['accept']	

	Set to a string or javascript array of strings describing CSS classes that the droppable element will accept. The drop element will only accept elements of the specified CSS classes.
$options['containment']	The droppable element will only accept the dragged element if it is contained in the given elements (element ids). Can be a string or a javascript array of id references.
$options['overlap']	If set to 'horizontal' or 'vertical', the droppable element will only react to a draggable element if it is overlapping the droparea by more than 50% in the given axis.
$options['onDrop']	A javascript call back that is called when the dragged element is dropped on the droppable element.

`dropRemote(string $id, array $options)`

Makes a drop target that creates an XMLHttpRequest when a draggable element is dropped on it. The $options array for this function are the same as those specified for drop() and link().

7.1.2.11 slider

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`slider(string $id, string $track_id, array $options)`

Creates a directional slider control. For more information see http://wiki.github.com/madrobby/scriptaculous/slider.

Common options might include:

$options keys	Description
$options['axis']	Sets the direction the slider will move in. 'horizontal' or 'vertical'. Defaults to horizontal
$options['handleImage']	The id of the image that represents the handle. This is used to swap out the image src with disabled image src when the slider is enabled. Used in conjunction with handleDisabled.
$options['increment']	Sets the relationship of pixels to values. Setting to 1 will make each pixel adjust the slider value by one.
$options['handleDisabled']	The id of the image that represents the disabled handle. This is used to change the image src when the slider is disabled. Used in conjunction handleImage.
$options['change'] $options['onChange']	JavaScript callback fired when the slider has finished moving, or has its value changed. The callback function receives the slider's current value as a parameter.
$options['slide'] $options['onSlide']	JavaScript callback that is called whenever the slider is moved by dragging. It receives the slider's current value as a parameter.

7.1.2.12 editor

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`editor(string $id, string $url, array $options)`

Creates an in-place editor at DOM id. The supplied `$url` should be an action that is responsible for saving element data. For more information and demos see http://github.com/madrobby/scriptaculous/wikis/ajax-inplaceeditor.

Common options might include:

$options keys	Description
`$options['collection']`	Activate the 'collection' mode of in-place editing. $options['collection'] takes an array which is turned into options for the select. To learn more about collection see http://github.com/madrobby/scriptaculous/wikis/ajax-inplacecollectioneditor.
`$options['callback']`	A function to execute before the request is sent to the server. This can be used to format the information sent to the server. The signature is `function(form, value)`
`$options['okText']`	Text of the submit button in edit mode
`$options['cancelText']`	The text of the link that cancels editing
`$options['savingText']`	The text shown while the text is sent to the server
`$options['formId']`	
`$options['externalControl']`	
`$options['rows']`	The row height of the input field
`$options['cols']`	The number of columns the text area should span
`$options['size']`	Synonym for 'cols' when using single-line
`$options['highlightcolor']`	The highlight color
`$options['highlightendcolor']`	The color which the highlight fades to
`$options['savingClassName']`	
`$options['formClassName']`	
`$options['loadingText']`	
`$options['loadTextURL']`	

Example

```
<div id="in_place_editor_id">Text To Edit</div>
<?php
echo $ajax->editor(
    "in_place_editor_id",
    array(
        'controller' => 'Posts',
        'action' => 'update_title',
        $id
    ),
```

```
    array()
);
?>
```

7.1.2.13 sortable

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`sortable(string $id, array $options)`

Makes a list or group of floated objects contained by $id sortable. The options array supports a number of parameters. To find out more about sortable see http://wiki.github.com/madrobby/scriptaculous/sortable.

Common options might include:

$options keys	Description
$options['tag']	Indicates what kind of child elements of the container will be made sortable. Defaults to 'li'.
$options['only']	Allows for further filtering of child elements. Accepts a CSS class.
$options['overlap']	Either 'vertical' or 'horizontal'. Defaults to vertical.
$options['constraint']	Restrict the movement of the draggable elements. accepts 'horizontal' or 'vertical'. Defaults to vertical.
$options['handle']	Makes the created Draggables use handles, see the handle option on Draggables.
$options['onUpdate']	Called when the drag ends and the Sortable's order is changed in any way. When dragging from one Sortable to another, the callback is called once on each Sortable.
$options['hoverclass']	Give the created droppable a hoverclass.
$options['ghosting']	If set to true, dragged elements of the sortable will be cloned and appear as a ghost, instead of directly manipulating the original element.

7.2 Cache

L'assistant Cache permet de mettre en cache des vues et layouts, ce qui permet de sauver le temps de récupération des données répétitives. Le système de cache des vues de Cake sauvegarde les vues et layouts avec le moteur de stockage de votre choix. Il faut noter que l'assistant Cache fonctionne de façon assez différente des autres assistants. 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.

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 sauve véritablement du temps pour chaque requête d'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.

7.2.1 Généralités sur la mise en cache

Mettre en cache permet un stockage temporaire pour aider à réduire les chargements sur le serveur. Par exemple, il est possible de stocker les résultats des requêtes coÃ»teuses en temps sur la base de données, de

manière à ne pas les demander pour chaque page.

Gardons ça à l'esprit, le cache n'est pas un stockage permanent et ne devrait jamais être utilisé à cette fin. Il faut mettre en cache des choses qu'on est capable de générer de nouveau au besoin.

7.2.2 Les moteurs de cache de Cake

Nouveauté depuis la version 1.2 : les moteurs de cache. Ce sont des interfaces transparentes avec l'assistant Cache qui vous autorisent à stocker des vues de différentes façons sans vous préoccuper du comment. Le choix du moteur de cache est positionné dans le fichier de configuration app/config/core.php. La plupart des options sont listées dans la version du fichier qui vient avec la distribution et des informations plus détaillées sur chaque moteur se trouve dans la section Cache.

Fichiers	Le moteur de cache de type fichier est celui que Cake utilise par défaut. Il écrit des fichiers plats dans l'arborescence de fichier, il possède des options mais fonctionne bien avec les options par défaut.
APC	Le moteur de cache APC implémente l'alternative APC : Alternative PHP Cache, le cache d'opcode libre et ouvert pour PHP. Comme XCache, ce moteur met en cache le code intermédiaire PHP.
XCache	Ce moteur de cache est fonctionnellement similaire à APC met implémente XCache, le cache d'opcode. Il nécessite de rentrer une authentification utilisateur pour fonctionner correctement.
Memcache	Le moteur de cache Memcache fonctionne avec serveur Memcache: un démon vous autorisant à créer un objet cache dans la mémoire vive du système. Plus d'information sur le module Memcache qui sert d'interface au démon sur php.net et sur le démon memcached ici.

7.2.3 La configuration de l'assistant Cache

La mise en cache de vue et l'Assistant Cache possèdent plusieurs éléments de configuration détaillés ci-dessous.

Pour utiliser l'Assistant Cache dans n'importe quel vue ou contrôleur, vous devez d'abord dé-commenter et définir Configure::Cache.check à true à la ligne 80 de `core.php` dans votre /app/config. Si ce n'est pas le cas, alors le cache ne sera pas vérifié, ni créé.

7.2.4 Mettre en cache depuis le contrôleur

Tout contrôleur qui utilise des fonctionnalités de cache doit inclure l'assistant cache (CacheHelper) dans son tableau d'assistants ($helpers).

```
var $helpers = array('Cache');
```

Vous devez aussi indiquer quelles actions doivent être mises en cache et pour combien de temps. On fait cela avec la variable $cacheAction du contrôleur. $cacheAction devrait être un tableau qui contient les actions concernées et la durée de mise en cache exprimée en seconde. La durée peut aussi être exprimé au format de la fonction strtotime() (ie : "1 hour", ou "3 minutes").

En utilisant l'exemple d'un contrôleur ArticlesController, qui reçoit beaucoup de trafic qu'on a besoin de mettre en cache.

Mise en cache des articles fréquemment lus pour des durées de temps variables :

```
var $cacheAction = array(
    'view/23/' => 21600,
    'view/48/' => 36000,
    'view/52'  => 48000
);
```

Mise en cache d'une action complète, dans notre cas un listing d'articles :

```
var $cacheAction = array(
    'archives/' => '60000'
);
```

Mise en cache de toutes les actions du contrôleur en utilisant un format strtotime() pour indiquer un temps long.

```
var $cacheAction = "1 hour";
```

Vous pouvez aussi activer les callbacks des contrôleurs/composants pour les vues mises en cache et créées avec le `CacheHelper`. Pour faire çà, vous devez utiliser le format tableau pour `$cacheAction` et créer un tableau comme celui-ci :

```
var $cacheAction = array(
    'view' => array('callbacks' => true, 'duration' => 21600),
    'add' => array('callbacks' => true, 'duration' => 36000),
    'index'  => array('callbacks' => true, 'duration' => 48000)
);
```

En définissant `callbacks => true`, vous indiquez à l'assistant Cache que vous voulez les fichiers générés pour créer les composants et modèles du contrôleur. En plus, cela lance les méthodes initialize et startup du composant, le beforeFilter du contrôleur.

Définir `callbacks => true` fait échouer en partie l'objectif de la mise en cache. C'est la raison pour laquelle ceci est désactivé par défaut.

7.2.5 Contenus non mis en cache 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 `<cake:nocache>` `</cake:nocache>` comme ci-dessous :

```
<cake:nocache>
<?php if ($session->check('Utilisateur.nom')) : ?>
    Bienvenue, <?php echo $session->read('Utilisateur.nom')?>.
<?php else: ?>
    <?php echo $html->link('Login', 'users/login')?>
<?php endif; ?>
</cake:nocache>
```

Il est à noter, qu'une fois une action mise en cache, la méthode du contrôleur correspondante ne sera plus appelée - sinon il n'y aurait pas d'intérêt à mettre la page en cache. Par conséquent, ce n'est pas possible d'entourer par `<cake:nocache> </cake:nocache>`, des variables qui ont été définies dans le contrôleur, puisqu'elles auront la valeur *null*.

7.2.6 Nettoyer le cache

Il est important de se rappeler que Cake va nettoyer le cache si un modèle utilisé dans la vue mise en cache a été modifié. Par exemple, si une vue mise en cache utilise des données du modèle Post et qu'il y a eu une requête INSERT, UPDATE, ou DELETE sur Post, le cache est nettoyé, et un nouveau contenu est généré à la prochaine requête.

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`

7.3 Forms

La première méthode dont vous aurez besoin d'utiliser pour prendre pleinement avantage du *FormHelper* est `create()`. Cette méthode affichera un tag d'ouverture de formulaire.

`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 contrôleur courant, avec selon les cas l'action `add()` ou l'action `edit()`. La méthode par défaut pour les formulaires est *POST*. L'élément du formulaire est également renvoyée avec un DOM ID. Cet identifiant est créé à partir du nom du modèle, et du nom du contrôleur. Si j'apelle `create()` dans une vue de MembresController, j'obtiendrai ce genre de rendu dans ma vue :

```
<form id="MembreAddForm" method="post" action="/membres/add">
```

La méthode `create()` nous permet également de personnaliser plusieurs paramètres. Premièrement, vous pouvez spécifier un nom de modèle. Ce faisant, vous modifiez le contexte de ce formulaire. Tous les champs seront supposés dépendre de ce modèle (sauf si spécifié), et tous les modèles devront être liés à lui. Si vous ne spécifiez pas de modèle, CakePHP supposera que vous utilisez le modèle par défaut pour le contrôleur courant.

```
<?php echo $form->create('Recette'); ?>

// affichera :
<form id="RecetteAddForm" method="post" action="/recettes/add">
```

Ce formulaire enverra les données à votre action `add()` de *RecettesController*. Cependant, vous pouvez utiliser la même logique pour créer et modifier des formulaires. Le helper *FormHelper* utiliser la propriété `$this->data` pour détecter automatiquement s'il faut créer un formulaire d'ajout ou de modification. Si `$this->data` contient un élément tabulaire après le nom du modèle, et que ce tableau contient une valeur non nulle pour la clé primaire du modèle, alors le *FormHelper* créera un formulaire de modification pour cet enregistrement précis. Par exemple, si on va à l'adresse **`http://site.com/recettes/edit/5`**, nous devrions obtenir ceci :

```
// controllers/recettes_controller.php:
<?php
function edit($id = null) {
    if (empty($this->data)) {
        $this->data = $this->Recette->findById($id);
    } else {
        // Le code de sauvegarde vient ici
    }
}
?>

// views/recettes/edit.ctp:

// Comme $this->data['Recipe']['id'] = 5, on doit obtenir un formulaire de
modification
<?php echo $form->create('Recipe'); ?>

//affichera :
<form id="RecetteEditForm" method="post" action="/recettes/edit/5">
<input type="hidden" name="_method" value="PUT" />
```

Comme c'est un formulaire de modification, un champ caché (*hidden*) est créé pour surcharger la méthode HTTP par défaut

Le tableau `$options` est l'endroit où la plupart des paramètres de configurations est stockée. Ce tableau peut contenir un certain nombre de paires clé-valeur qui peuvent affecter la manière dont le formulaire sera créé.

7.3.1.1 $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.

```
<?php echo $form->create('Membre', array('type' => 'get')); ?>

//Affichera :
<form id="MembreAddForm" method="get" action="/membres/add">
```

Choisir 'file' changera la méthode de soumission à 'post', et ajoutera la mention "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 d'enctype empêchera l'envoi de fichiers à partir de ce formulaire.

```
<?php echo $form->create('Membre', array('type' => 'file')); ?>

//Affichera :
<form id="MembreAddForm" enctype="multipart/form-data" method="post"
action="/membres/add">
```

Quand vous utilisez 'put' ou 'delete', votre formulaire sera équivalent à un formulaire de type 'post', mais quand il sera envoyé, la méthode de requête HTTP sera respectivement surchargée avec 'PUT' ou 'DELETE'. Ca permettra à CakePHP de créer son propre support REST dans les navigateurs web.

7.3.1.2 $options['action']

La variable action vous permet de définir à quelle action dans votre contrôleur pointera le formulaire. Par exemple, si vous voulez que le formulaire appelle l'action login() de votre contrôleur courant, vous créeriez le tableau $options comme ceci :

```
<?php echo $form->create('Membre', array('action' => 'login')); ?>

//Affichera :
<form id="MembreLoginForm" method="post" action="/membres/login">
</form>
```

7.3.1.3 $options['url']

Si l'action que vous désirez appeler avec le formulaire n'est pas dans le contrôleur courant, vous pouvez spécifier une URL précise dans le formulaire en utilisant la clé 'url' de votre tableau $options. L'URL ainsi donnée peut être relative à votre application CakePHP ou peut pointer vers un domaine extérieur.

```
<?php echo $form->create(null, array('url' => '/recettes/ajouter')); ?>
// ou
<?php echo $form->create(null, array('url' => array('controller' =>
'recettes', 'action' => 'ajouter'))); ?>


// Affichera :
<form method="post" action="/recettes/ajouter">

<?php echo $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 différent types d'urls.

7.3.1.4 $options['default']

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

You can declare a set of default options for `input()` with the `inputDefaults` key to customize your default input creation.

```
echo $this->Form->create('User', array(
        'inputDefaults' => array(
            'label' => false,
            'div'   => false,
            # define error defaults for the form
            'error' => array(
              'wrap'  => 'span',
              'class' => 'my-error-class'
            )
        )
    ));
```

All inputs created from that point forward would inherit the options declared in inputDefaults. You can override the defaultOptions by declaring the option in the input() call.

```
echo $this->Form->input('password'); // No div, no label
echo $this->Form->input('username', array('label' => 'Username')); // has a label
element
```

7.3.2 Fermeture du Formulaire

Le FormHelper inclus également une méthode end() qui complète le marquage du formulaire. Souvent, end() affiche juste la base fermante du formulaire, mais le FormHelper permet aussi d'ajouter des champs cachées en utilisant la méthode end() other methods may be depending on.

```
<?php echo $form->create(); ?>

<!-- Form elements go here -->

<?php echo $form->end(); ?>
```

Si une chaine 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.

```
<?php echo $form->end('Finish'); ?>

Sortie :

<div class="submit">
    <input type="submit" value="Finish" />
</div>
</form>
```

7.3.3 Éléments de formulaire automagique

Tout d'abord, intéressons-nous à quelques unes des méthodes de création automatique de formulaire de l'assistant Forms. La principale méthode que nous allons étudier est input(). Cette méthode inspecte automatiquement le champ du modèle qui lui est fourni afin de créer une entrée appropriée pour ce champ.

`input(string $fieldName, array $options = array())`

Type de champ SQL	Champ du formulaire
string (char, varchar, etc.)	text
boolean, tinyint(1)	checkbox
text	textarea
text, with name of password, passwd, or psword	password
date	day, month, and year selects
datetime, timestamp	day, month, year, hour, minute, and meridian selects
time	hour, minute, and meridian selects

Par exemple, supposons que mon modèle Utilisateur contient les champs nom_utilisateur (varchar), mot_de_passe (varchar), approuve (datetime) et citation (text). Je peut utiliser la méthode input() de l'assistant Forms pour créer une entrée appropriée pour tous les champs du formulaire.

```
<?php echo $form->create(); ?>

    <?php
        echo $form->input('nom_utilisateur');    //text
        echo $form->input('mot_de_passe');    //password
        echo $form->input('approuve');    //day, month, year, hour, minute, meridian
        echo $form->input('citation');       //textarea
    ?>

<?php echo $form->end('Add'); ?>
```

Un exemple plus complet montrant quelques options pour le champ de date :

```
        echo $form->input('date_naissance', array( 'label' => 'Date de naissance'
                                    , 'dateFormat' => 'DMY'
                                    , 'minYear' => date('Y') - 70
                                    , 'maxYear' => date('Y') - 18 ));
```

Et pour finir, voici un exemple pour la création d'une selection hasAndBelongsToMany. Supposons que Utilisateur hasAndBelongsToMany Groupe. Dans votre controlleur, définissez une variable camelCase au pluriel (groupe -> groupes dans cette exemple, ou ExtraFunkyModele -> extraFunkyModeles) avec les options de sélection. Dans le controlleur vous pouver définir :

```
$this->set('groupes', $this->Utilisateur->Groupe->find('list'));
```

Et dans la vue une sélection multiple sera créée avec cette simple ligne de code :

```
echo $form->input('Groupe');
```

Si vous voulez un champ de sélection utilisant une relation belongsTo ou hasOne, vous pouvez ajouter ceci dans votre controlleur Utilisateurs (supposant que Utilisateur belongsTo Groupe):

```
$this->set('groupes', $this->Utilisateur->Groupe->find('list'));
```

Ensuite, ajoutez ceci à la vue du formulaire :

```
echo $form->input('groupe_id');
```

7.3.3.1 Convention 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 modèle courant, selon le format

suivant :

```
<input type="text" id="NommodeleNomchamp" name="data[Nommodele][nomchamp]">
```

Vous pouvez également préciser le nom du modèle manuellement, en passant un premier paramètre de la forme Nommodele.nomchamp.

```
echo $form->input('Nommodele.nomchamp');
```

Si vous avez besoin de définir plusieurs champs ayant le même nom, donc de créer un tableau qui peut être enregistré en une seule fois avec `saveAll()`, utilisez la convention suivante :

```
<?php
   echo $form->input('Nommodele.0.nomchamp');
   echo $form->input('Nommodele.1.nomchamp');
?>

<input type="text" id="Nommodele0Nomchamp" name="data[Nommodele][0][nomchamp]">
<input type="text" id="Nommodele1Nomchamp" name="data[Nommodele][1][nomchamp]">
```

7.3.3.2 $options['type']

Vous pouvez forcer le type d'un input (et donc remplacer la logique d'analyse du modèle) en définissant un type. En plus des types de champs décrits dans le tableau ci-dessus, vous pouvez également créer des inputs 'file' et 'password'.

```
<?php echo $form->input('champ', array('type' => 'file')); ?>

Affiche :

<div class="input">
    <label for="UtilisateurChamp">Champ</label>
    <input type="file" name="data[Utilisateur][champ]" value="" id="UtilisateurChamp"
/>
</div>
```

7.3.3.3 $options['before'], $options['between'], $options['separator'] and $options['after']

Utilisez ces clés si vous avez besoin d'injecter quelques balises à la sortie de la méthode input().

```
<?php echo $form->input('field', array(
    'before' => '--avant--',
    'after' => '--après--',
    'between' => '--au milieu---'
));?>

Output:

<div class="input">
--avant--
<label for="UserField">Champ</label>
--au milieu---
<input name="data[User][field]" type="text" value="" id="UserField" />
```

```
--après--
</div>
```

Pour un *input* de type radio l'attribut *'separator'* peut être utilisé pour injecter des balise pour séparer input/label.

```
<?php echo $form->input('field', array(
    'before' => '--avant--',
    'after' => '--après--',
    'between' => '--au milieu--',
    'separator' => '--séparateur--',
    'options' => array('1', '2')
));?>

Output:

<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>
--au milieu---
--après--
</div>
```

Pour un élément de type `date` et `datetime` l'attribut *'separator'* peut être utilisé pour modifier la chaine entre les *select*. Par défaut '-'.

7.3.3.4 $options['options']

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

This key allows you to manually specify options for a select input, or for a radio group. Unless the 'type' is specified as 'radio', the FormHelper will assume that the target output is a select input.

```
<?php echo $this->Form->input('field', array('options' => array(1,2,3,4,5)));
?>
```

Output:

```
<div class="input">
    <label for="UserField">Field</label>
    <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>
</div>
```

Options can also be supplied as key-value pairs.

```
<?php echo $this->Form->input('field', array('options' => array(
    'Value 1'=>'Label 1',
    'Value 2'=>'Label 2',
    'Value 3'=>'Label 3'
 ))); ?>
```

Output:

```
<div class="input">
    <label for="UserField">Field</label>
    <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>
</div>
```

If you would like to generate a select with optgroups, just pass data in hierarchical format. Works on multiple checkboxes and radio buttons too, but instead of optgroups wraps elements in fieldsets.

```
<?php echo $this->Form->input('field', array('options' => array(
    'Label1' => array(
       'Value 1'=>'Label 1',
       'Value 2'=>'Label 2'
    ),
    'Label2' => array(
       'Value 3'=>'Label 3'
    )
 ))); ?>
```

Output:

```
<div class="input">
    <label for="UserField">Field</label>
    <select name="data[User][field]" id="UserField">
        <optgroup label="Label1">
            <option value="Value 1">Label 1</option>
            <option value="Value 2">Label 2</option>
        </optgroup>
        <optgroup label="Label2">
            <option value="Value 3">Label 3</option>
        </optgroup>
    </select>
</div>
```

7.3.3.5 $options['multiple']

Si 'multiple' a été définit à vrai pour un champ qui génère un select, le select autorisera les sélections multiples. Il est également possible de définir la valeur de l'option 'multiple' à 'checkbox' pour générer une liste de case à cocher.

```
$form->input('Model.field', array( 'type' => 'select', 'multiple' => true ));
```

```
$form->input('Model.field', array( 'type' => 'select', 'multiple' => 'checkbox'
));
```

7.3.3.6 $options['maxLength']

utiliser cette option pour mettre a jour les attributs contenus dans la balise div. L'introduction d'une chaine de caractère mettra a jour
l'attribut class . L'introduction d'un tableau mettra a jour les attributs correspondants au champs clé/valeur du tableau . Alternativement ,
vous pouvez mettre cette option a faux pour pour annuler l'affichage du div .

Modification du nom de la class :

```
    echo $form->input('User.name', array('div' => 'class_name'));
```

Code produit :

```
<div class="class_name">
        <label for="UserName">Name</label>
        <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Modification de plusieurs attributs :

```
    echo $form->input('User.name', array('div' => array('id' => 'mainDiv',
'title' => 'Div Title', 'style' => 'display:block')));
```

Code produit :

```
<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>
```

Annulation de l'affichage du div :

```
    <?php echo $form->input('User.name', array('div' => false));?>
```

Code produit :

```
    <label for="UserName">Name</label>
    <input name="data[User][name]" type="text" value="" id="UserName" />
```

7.3.3.8 $options['label']

Mettez a jour cette option pour modifier la chaine de caractere qui va etre affichée dans le libellé qui va accompagner le input

```
<?php echo $form->input( 'User.name', array( 'label' => 'The User Alias' )
);?>
```

Code produit :

```
<div class="input">
    <label for="UserName">The User Alias</label>
    <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

Alternativement , mettez cette option a faux pour annuler l'affichage du libellé .

```
<?php echo $form->input( 'User.name', array( 'label' => false ) ); ?>
```

Code produit :

```
<div class="input">
    <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

mettez cette option sous forme de tableau pour apporter des options supplémentaires a l'élément `label` . Si vous faites cela , vous pourrai utiliser la clé `text` dans le tableau pour modifier le texte du libellé .

```
<?php echo $form->input( 'User.name', array( 'label' => array('class' =>
'thingy', 'text' => 'The User Alias') ) ); ?>
```

Code produit :

```
<div class="input">
    <label for="UserName" class="thingy">The User Alias</label>
    <input name="data[User][name]" type="text" value="" id="UserName" />
</div>
```

7.3.3.9 $options['legend']

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Using this key allows you to override the default model error messages and can be used, for example, to set i18n messages. It has a number of suboptions which control the wrapping element, wrapping element class name, and whether HTML in the error message will be escaped.

To disable error message output set the error key to false.

```
$this->Form->input('Model.field', array('error' => false));
```

To modify the wrapping element type and its class, use the following format:

```
$this->Form->input('Model.field', array('error' => array('wrap' => 'span',
'class' => 'bzzz')));
```

To prevent HTML being automatically escaped in the error message output, set the escape suboption to false:

```
$this->Form->input('Model.field', array('error' => array('escape' => false)));
```

To override the model error messages use an associate array with the keyname of the validation rule:

```
$this->Form->input('Model.field', array('error' => array('tooShort' =>
__('This is not long enough', true) )));
```

As seen above you can set the error message for each validation rule you have in your models. In addition you can provide i18n messages for your forms.

7.3.3.12 $options['default']

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Used to set a default value for the input field. The value is used if the data passed to the form does not contain a value for the field (or if no data is passed at all).

Example usage:

```
<?php
    echo $this->Form->input('ingredient', array('default'=>'Sugar'));
?>
```

Example with select field (Size "Medium" will be selected as default):

```
<?php
    $sizes = array('s'=>'Small', 'm'=>'Medium', 'l'=>'Large');
    echo $this->Form->input('size', array('options'=>$sizes, 'default'=>'m'));

?>
```

You cannot use `default` to check a checkbox - instead you might set the value in `$this->data` in your controller, `$this->Form->data` in your view, or set the input option `checked` to true.

Date and datetime fields' default values can be set by using the 'selected' key.

7.3.3.13 $options['selected']

Utilisé en combinaison avec un *input* de type *select* (A savoir: pour les types *select*, *date*, *time*, *datetime*). Défini la valeur *'selected'* de l'élément que vous voulez sélectionner par défaut lors de l'affichage de l'*input*.

```
echo $form->input('close_time', array('type' => 'time', 'selected' =>
'13:30:00'));
```

La clé sélectionnée pour un *input* de type *date* et *datetime* peut être un *timestamp* UNIX.

7.3.3.14 $options['rows'], $options['cols']

Ces deux clés définissent le nombre de lignes et de colonnes dans un *input* de type *textarea*.

```
echo $form->input('textarea', array('rows' => '5', 'cols' => '5'));
```

Affichera:

```
<div class="input text">
    <label for="FormTextarea">Textarea</label>
    <textarea name="data[Form][textarea]" cols="5" rows="5" id="FormTextarea" >
    </textarea>
</div>
```

7.3.3.15 $options['empty']

Si vrai, force l'*input* à rester vide.

Lorsqu'il est passé à une liste de sélection, il créé une option vide avec une valeur vide dans votre liste déroulante. Si vous voulez avoir une valeur vide avec un texte affiché au lieu d'une option vide, passer lui une chaine.

```
<?php echo $form->input('field', array('options' => array(1,2,3,4,5), 'empty'
=> '(choisissez un texte)')); ?>
```

Affichera:

```
<div class="input">
    <label for="UserField">Field</label>
    <select name="data[User][field]" id="UserField">
        <option value="">(choisissez un text)</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>
```

Si vous avez besoin de définir une valeur par défaut dans un champ *password* à vide, utiliser à la place 'value' => ''.

Les options peuvent être fournies sous forme de paire clés-valeurs.

7.3.3.16 $options['timeFormat']

Cette option spécifie le nombre de minutes entre chaque option dans la boîte de sélection des minutes.

```
<?php echo $form->input('Model.time', array('type' => 'time', 'interval' =>
15)); ?>
```

Créera 4 options dans la boite de sélection des minutes. Une toute les 15 minutes.

7.3.3.20 $options['class']

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

You can set the classname for an input field using `$options['class']`

```
echo $this->Form->input('title', array('class' => 'custom-class'));
```

7.3.3.21 $options['hiddenField']

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

For certain input types (checkboxes, radios) a hidden input is created so that the key in $this->data will exist even without a value specified.

```
<input type="hidden" name="data[Post][Published]" id="PostPublished_" value="0" />
<input type="checkbox" name="data[Post][Published]" value="1" id="PostPublished" />
```

This can be disabled by setting the `$options['hiddenField'] = false`.

```
echo $this->Form->checkbox('published', array('hiddenField' => false));
```

Which outputs:

```
<input type="checkbox" name="data[Post][Published]" value="1" id="PostPublished" />
```

If you want to create multiple blocks of inputs on a form that are all grouped together, you should use this parameter on all inputs except the first. If the hidden input is on the page in multiple places, only the last group of input's values will be saved

In this example, only the tertiary colors would be passed, and the primary colors would be overridden

```
<h2>Primary Colors</h2>
<input type="hidden" name="data[Color][Color]" id="Colors_" value="0" />
<input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsRed" />
<label for="ColorsRed">Red</label>
<input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsBlue" />
<label for="ColorsBlue">Blue</label>
<input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsYellow" />
<label for="ColorsYellow">Yellow</label>

<h2>Tertiary Colors</h2>
<input type="hidden" name="data[Color][Color]" id="Colors_" value="0" />
<input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsGreen" />
<label for="ColorsGreen">Green</label>
<input type="checkbox" name="data[Color][Color][]" value="5" id="ColorsPurple" />
<label for="ColorsPurple">Purple</label>
<input type="checkbox" name="data[Addon][Addon][]" value="5" id="ColorsOrange" />
<label for="ColorsOrange">Orange</label>
```

Disabling the `'hiddenField'` on the second input group would prevent this behavior

7.3.4 Champs de fichiers

Pour ajouter un champ d'*upload* de fichier dans un formulaire, vous devez d'abord vous assurer que l'attribut enctype du formulaire est fixé à "multipart/form-data", vous devez donc commencer par une fonction de création définie comme ci-dessous.

```
echo $this->Form->create('Document', array('enctype' => 'multipart/form-data')
);

// ou

echo $this->Form->create('Document', array('type' => 'file'));
```

Ensuite, ajoutez une des deux lignes suivantes à votre fichier de vue formulaire.

```
echo $this->Form->input('Document.fichiersoumis', array('between'=>'<br
/>','type'=>'file'));

// ou

echo $this->Form->file('Document.fichiersoumis');
```

A cause des limitations liées à HTML, il n'est pas possible de définir une valeur par défaut dans les champs inputs de type 'file'. Chaque fois que le formulaire est affiché, le champ sera vide.

Dès la soumission, les champs de fichier fournissent un tableau étendu de données au script qui reçoit les données du formulaire.

Dans l'exemple ci-dessus, les valeurs du tableau de données soumis seraient organisées de la manière suivante, si CakePHP était installé sur un serveur Windows. 'tmp_name' aurait un chemin différent dans un environnement Unix.

```
$this->data['Document']['fichiersoumis'] = array(
    'name' => planning_conference.pdf
    'type' => application/pdf
    'tmp_name' => C:/WINDOWS/TEMP/php1EE.tmp
    'error' => 0
    'size' => 41737
);
```

Ce tableau est généré par PHP lui-même, donc pour plus de détail sur la façon dont PHP gère les données passées dans les champ de fichier, lisez la section sur l'upload de fichier du manuel PHP.

7.3.4.1 Valider un upload de fichier

Voici un exemple de méthode de validation que vous pourriez définir dans votre modèle, afin de vérifier qu'un fichier a été uploadé avec succès.

```
// Basé sur le commentaire 8 de :
http://bakery.cakephp.org/articles/view/improved-advance-validation-with-parameters

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']);
    } else {
        return false;
    }
}
```

7.3.5 Eléments du Formulaire - Méthodes Spécifiques

Les autres méthodes disponibles dans l'Assistant 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).

```
<?php echo $form->text('pseudo', array('class' => 'utilisateurs')); ?>

Affichera :

<input name="data[Utilisateur][pseudo]" type="text" class="utilisateurs"
id="UtilisateurPseudo" />
```

7.3.5.1 checkbox

`checkbox(string $fieldName, array $options)`

Cette méthode créer une checkbox. Elle génère également un champ input de type hidden afin de forcer la soumission des données pour le champ spécifié.

```
<?php echo $this->Form->checkbox('fait'); ?>
```

Donnera:

```
<input type="hidden" name="data[Utilisateur][fait]" value="0" id="UtilisateurFait_" />
<input type="checkbox" name="data[Utilisateur][fait]" value="1" id="UtilisateurFait" />
```

7.3.5.2 button

button(string $title, array $options = array())

Crée un bouton HTML avec le titre spécifié et un type de *"button"* par défaut. La configuration de `$options['type']` affichera l'un des 3 types de bouton possible:

1. button: Créer un bouton standard (celui par défaut).
2. reset: Créer un bouton de réinitialisation de formulaire.
3. submit: Similaire à la methode `$form->submit`.

```
<?php
echo $form->button('Un bouton');
echo $form->button('Un autre  bouton', array('type'=>'button'));
echo $form->button('Réinitialiser le formulaire', array('type'=>'reset'));
echo $form->button('Soumettre le formulaire', array('type'=>'submit'));
?>
```

Devrai afficher:

```
<input type="button" value="Un bouton" />
<input type="button" value="Un autre bouton" />
<input type="reset" value="Réinitialiser le formulaire" />
<input type="Submit" value="Soumettre le formulaire" />
```

7.3.5.3 year

year(string $fieldName, int $minYear, int $maxYear, mixed $selected, array $attributes, boolean $showEmpty)

Crée un menu de sélection composé des années de `$minYear` à `$maxYear`, avec l'année $selected sélectionnée par défaut. Les attributs HTML sont fournit dans $attributes. Si `$showEmpty` est faux, le menu de sélection n'affichera pas de champs vide.

```
<?php
echo $form->year('purchased',2000,date('Y'));
?>
```

Affichera:

```
<select name="data[User][purchased][year]" 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>
```

7.3.5.4 month

month(string $fieldName, mixed $selected, array $attributes, boolean $showEmpty)

Crée un menu de sélection composé des noms des mois.

```
<?php
echo $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>
```

7.3.5.5 dateTime

dateTime(string $fieldName, string $dateFormat = 'DMY', $timeFormat = '12', mixed $selected = null, array $attributes = array())

Crée un menu de sélection pour la date et le temps. Les valeurs valides de $dateformat sont 'DMY', 'MDY', 'YMD' ou 'NONE'. Les valeurs valides pour $timeFormat sont '12', '24', et 'NONE'.

Vous pouvez préciser de ne pas afficher de valeur vide, en spécifiant "array('empty' => false)" dans le paramètre 'attributes'.
Vous pouvez aussi pré-sélectionner la date (et l'heure) courante en mettant $selected = null et $attributes = array("empty" => false).

7.3.5.6 day

day(string $fieldName, mixed $selected, array $attributes, boolean $showEmpty)

Crée un menu de sélection composé des jours (numériques) du mois.

Pour créer une option vide avec un texte de votre choix (par exemple, la première option est "Jour"), vous pouvez définir le texte comme paramètre final:

```
<?php
echo $form->day('created');
?>
```

Devrai afficher:

```
<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>
```

7.3.5.7 hour

hour(string $fieldName, boolean $format24Hours, mixed $selected, array $attributes, boolean $showEmpty)

Crée un menu de sélection composé de l'heure du jour.

7.3.5.8 minute

minute(string $fieldName, mixed $selected, array $attributes, boolean $showEmpty)

Crée un menu de sélection composé des minutes de l'heure.

7.3.5.9 meridian

meridian(string $fieldName, mixed $selected, array $attributes, boolean $showEmpty)

Crée un menu de sélection composé de 'am' et 'pm'.

7.3.5.10 error

error(string $fieldName, string $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' bool Échapper ou non le contenu de l'erreur.
- 'wrap' mixed Enveloppe ou non le message d'erreur d'une div. Si c'est une chaine, elle sera utilisé comme tag HTML.
- 'class' string Le nom de la *class* du message d'erreur

7.3.5.11 file

file(string $fieldName, array $options)

Crée un *input* de type *file*.

```
<?php
echo $form->create('User',array('type'=>'file'));
echo $form->file('avatar');
?>
```

Devrai afficher:

```
<form enctype="multipart/form-data" method="post" action="/users/add">
<input name="data[User][avatar]" value="" id="UserAvatar" type="file">
```

Lors de l'utilisation de `$form->file()`, rappelez vous de bien utilisé l'*encoding-type file*, en définissant le type en option à 'file' dans `$form->create()`

7.3.5.12 hidden

`hidden(string $fieldName, array $options)`

Crée un champs invisible. Exemple:

```
<?php
echo $form->hidden('id');
?>
```

Devrai afficher:

```
<input name="data[User][id]" value="10" id="UserId" type="hidden">
```

7.3.5.13 isFieldError

`isFieldError(string $fieldName)`

Renvoie vrai si le champ $fieldName a une erreur de validation.

```
<?php
if ($form->isFieldError('genre')){
    echo $form->error('genre');
}
?>
```

Lors de l'utilisation de `$form->input()`, les erreurs sont affichées par défaut.

7.3.5.14 label

`label(string $fieldName, string $text, array $attributes)`

Crée une étiquette (*tag label*), contenant $text.

```
<?php
echo $form->label('status');
?>
```

Devrai afficher:

```
<label for="UserStatus">Status</label>
```

7.3.5.15 password

password(string $fieldName, array $options)

Crée un champs de mot de passe.

```
<?php
echo $form->password('password');
?>
```

Devrai afficher:

```
<input name="data[User][password]" value="" id="UserPassword" type="password">
```

7.3.5.16 radio

radio(string $fieldName, array $options, array $attributes)

Crée un bouton de type *radio*. Utilisez `$attributes['value']` pour définir quelle valeur devra être sélectionnée par défaut.

Utilisez `$attributes['separator']` pour spécifier le HTML entre les boutons *radio* (e.g.
).

Les boutons sont enveloppés par défaut d'un *label* et d'un *fieldset*. Définissez `$attributes['legend']` à *false* pour les supprimer.

```
<?php
$options=array('M'=>'Male','F'=>'Female');
$attributes=array('legend'=>false);
echo $form->radio('gender',$options,$attributes);
?>
```

Devrait afficher:

```
<input name="data[User][gender]" id="UserGender_" value="" type="hidden">
<input name="data[User][gender]" id="UserGenderM" value="M" type="radio">
<label for="UserGenderM">Male</label>
<input name="data[User][gender]" id="UserGenderF" value="F" type="radio">
<label for="UserGenderF">Female</label>
```

Si pour n'importe quelle raison vous ne voulez pas de l'*input* caché, définissez `$attributes['value']` par une valeur à sélectionner ou un booléen à *false*.

7.3.5.17 select

select(string $fieldName, array $options, mixed $selected, array $attributes)

Crée un menu de sélection, composé des éléments de `$options`, avec l'option spécifiée par `$selected` qui sera le champ sélectionné par défaut. Pour afficher votre propre option par défaut, assignez la valeur souhaitée à `$attributes['empty']` ou fixez sa valeur à *false* pour désactiver l'option vide.

```
<?php
$options=array('M'=>'Homme','F'=>'Femme');
echo $form->select('sexe',$options)
?>
```

Devrai afficher:

```
<select name="data[User][sexe]" id="UserSexe">
<option value=""></option>
<option value="M">Homme</option>
<option value="F">Femme</option>
</select>
```

7.3.5.18 submit

`submit(string $caption, array $options)`

Crée un bouton de soumission de formulaire avec une légende `$caption`. Si `$caption` est l'URL d'image (qui contient un '.'), le bouton sera affiché en tant qu'image.

Il est enveloppé d'une `div` par défaut ; vous pouvez annuler cette déclaration `$options['div'] = false`.

```
<?php
echo $form->submit();
?>
```

Devrait afficher:

```
<div class="submit"><input value="Submit" type="submit"></div>
```

Vous pouvez définir une url d'image relative ou absolue pour la légende à la place d'un texte comme légende.

```
<?php
echo $form->submit('ok.png');
?>
```

Devrait afficher:

```
<div class="submit"><input type="image" src="/img/ok.png"></div>
```

7.3.5.19 text

`text(string $fieldName, array $options)`

Crée un champ de texte.

```
<?php
echo $form->text('prenom');
?>
```

Devrai afficher:

```
<input name="data[User][prenom]" value="" id="UserPrenom" type="text">
```

7.3.5.20 textarea

textarea(string $fieldName, array $options)

Crée un champ de zone de texte.

```
<?php
echo $form->textarea('notes');
?>
```

Devrai afficher:

```
<textarea name="data[User][notes]" id="UserNotes"></textarea>
```

7.3.6 1.3 improvements

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The FormHelper is one of the most frequently used classes in CakePHP, and has had several improvements made to it.

Entity depth limitations

In 1.2 there was a hard limit of 5 nested keys. This posed significant limitations on form input creation in some contexts. In 1.3 you can now create infinitely nested form element keys. Validation errors and value reading for arbitrary depths has also been added.

Model introspection

Support for adding 'required' classes, and properties like `maxlength` to hasMany and other associations has been improved. In the past only 1 model and a limited set of associations would be introspected. In 1.3 models are introspected as needed, providing validation and additional information such as maxlength.

Default options for input()

In the past if you needed to use `'div' => false`, or `'label' => false` you would need to set those options on each and every call to `input()`. Instead in 1.3 you can declare a set of default options for `input()` with the `inputDefaults` key.

```
echo $this->Form->create('User', array(
        'inputDefaults' => array(
            'label' => false,
```

```
            'div' => false
        )
    ));
```

All inputs created from that point forward would inherit the options declared in inputDefaults. You can override the defaultOptions by declaring the option in the input() call.

```
echo $this->Form->input('password'); // No div, no label
echo $this->Form->input('username', array('label' => 'Username')); // has a label
element
```

Omit attributes

You can now set any attribute key to null or false in an options/attributes array to omit that attribute from a particular html tag.

```
echo $this->Form->input('username', array(
    'div' => array('class' => false)
)); // Omits the 'class' attribute added by default to div tag
```

Accept-charset

Forms now get an accept-charset set automatically, it will match the value of `App.encoding`, it can be overridden or removed using the 'encoding' option when calling create().

```
// To remove the accept-charset attribute.
echo $this->Form->create('User', array('encoding' => null));
```

Removed parameters

Many methods such as `select`, `year`, `month`, `day`, `hour`, `minute`, `meridian` and `datetime` took a `$showEmpty` parameter, these have all been removed and rolled into the `$attributes` parameter using the `'empty'` key.

Default url

The default url for forms either was `add` or `edit` depending on whether or not a primary key was detected in the data array. In 1.3 the default url will be the current action, making the forms submit to the action you are currently on.

Disabling hidden inputs for radio and checkbox

The automatically generated hidden inputs for radio and checkbox inputs can be disabled by setting the `'hiddenField'` option to `false`.

button()

button() now creates button elements, these elements by default do not have html entity encoding enabled. You can enable html escaping using the `escape` option. The former features of `FormHelper::button` have been moved to `FormHelper::submit`.

submit()

Due to changes in `button()`, `submit()` can now generate reset, and other types of input buttons. Use the `type` option to change the default type of button generated. In addition to creating all types of buttons, `submit()` has `before` and `after` options that behave exactly like their counterparts in `input()`.

$options['format']

The HTML generated by the form helper is now more flexible than ever before. The $options parameter to Form::input() now supports an array of strings describing the template you would like said element to follow. It's just been recently added to SCM, and has a few bugs for non PHP 5.3 users, but should be quite useful for all. The supported array keys are `array('before', 'input', 'between', 'label', 'after', 'error')`.

7.4 HTML

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The role of the HtmlHelper in CakePHP is to make HTML-related options easier, faster, and more resilient to change. Using this helper will enable your application to be more light on its feet, and more flexible on where it is placed in relation to the root of a domain.

Before we look at HtmlHelper's methods, you'll need to know about a few configuration and usage situations that will help you use this class. First in an effort to assuage those who dislike short tags (<?= ?>) or many echo() calls in their view code all methods of HtmlHelper are passed to the output() method. If you wish to enable automatic output of the generated helper HTML you can simply implement output() in your AppHelper.

```
function output($str) {
    echo $str;
}
```

Doing this will remove the need to add echo statements to your view code.

Many HtmlHelper methods also include a $htmlAttributes parameter, that allow you to tack on any extra attributes on your tags. Here are a few examples of how to use the $htmlAttributes parameter:

```
Desired attributes: <tag class="someClass" />
Array parameter: array('class'=>'someClass')

Desired attributes: <tag name="foo" value="bar" />
Array parameter:  array('name' => 'foo', 'value' => 'bar')
```

The HtmlHelper is available in all views by default. If you're getting an error informing you that it isn't there, it's usually due to its name being missing from a manually configured $helpers controller variable.

7.4.1 Insérer des balises bien formatés

La tache la plus importante de l'assistant HTML est de créer des balises bien formatés. N'ayez pas peur de l'utiliser souvent - vous pouvez mettre en cache les Vues dans CakePHP pour économiser un peu de CPU lorsque les vues sont affichées. Cette section porte sur plusieurs des méthodes de l'assistant HTML et la façon de les utiliser.

7.4.1.1 charset

charset(string $charset=null)

Utilisé pour créer une balise META précisant le type d'encodage des caractères du document. Par défaut UTF-8.

```
<?php echo $this->Html->charset(); ?>
```

Va afficher:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Ainsi que,

```
<?php echo $this->Html->charset('ISO-8859-1'); ?>
```

Va afficher:

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
```

7.4.1.2 css

css(mixed $path, string $rel = null, array $options = array())

Crée un lien vers une feuille de style CSS. Si la clé 'inline' est définie à false dans le paramètre $options, les balises seront ajoutées à la variable $scripts_for_layout que vous pouvez intégrer à l'intérieur de la balise head du document.

Cette méthode d'inclusion suppose que le fichier spécifié réside à l'intérieur du dossier /app/webroot/css ...

```
<?php echo $this->Html->css('formulaires'); ?>
```

Affichera :

```
<link rel="stylesheet" type="text/css" href="/fr/css/formulaires.css" />
```

Le premier paramètre peut être un tableau pour inclure plusieurs fichiers.

```
<?php echo $this->Html->css(array('formulaires','tableaux','menu')); ?>
```

Affichera :

```
<link rel="stylesheet" type="text/css" href="/fr/css/formulaires.css" />
<link rel="stylesheet" type="text/css" href="/fr/css/tableaux.css" />
<link rel="stylesheet" type="text/css" href="/fr/css/menu.css" />
```

7.4.1.3 meta

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`meta(string $type, string $url = null, array $attributes = array())`

This method is handy for linking to external resources like RSS/Atom feeds and favicons. Like css(), you can specify whether or not you'd like this tag to appear inline or in the head tag by setting the 'inline' key in the $attributes parameter to false, ie - `array('inline' => false)`.

If you set the "type" attribute using the $attributes parameter, CakePHP contains a few shortcuts:

type	translated value
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) </p>
<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"
/>
```

This method can also be used to add the meta keywords and descriptions. Example:

```
<?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"/>
```

If you want to add a custom meta tag then the first parameter should be set to an array. To output a robots noindex tag use the following code:

```
 echo $this->Html->meta(array('name' => 'robots', 'content' => 'noindex'));
```

7.4.1.4 docType

docType(string $type = 'xhtml-strict')

Retourne une balise doctype (X)HTML. Les doctypes fournis sont ceux du tableau suivant:

type	valeur traduite
html	text/html
html4-strict	HTML4 Strict
html4-trans	HTML4 Transitional
html4-frame	HTML4 Frameset
xhtml-strict	XHTML1 Strict
xhtml-trans	XHTML1 Transitional
xhtml-frame	XHTML1 Frameset
xhtml11	XHTML 1.1

```
<?php echo $html->docType(); ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<?php echo $html->docType('html4-trans'); ?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

7.4.1.5 style

style(array $data, boolean $oneline = true)

Construit des définitions de styles CSS, basées sur les clés et valeurs du tableau passé à la méthode. Particulièrement pratique si votre fichier CSS est dynamique.

```
<?php echo $this->Html->style(array(
    'background'     => '#633',
    'border-bottom' => '1px solid #000',
    'padding' => '10px'
)); ?>
```

Affichera :

```
  background:#633;border-bottom:1px solid #000;padding:10px;
```

7.4.1.6 image

image(string $path, array $htmlAttributes = array())

Crée un tag d'image formaté. Le chemin fourni doit être relatif à **/app/webroot/img/**.

```
<?php echo $html->image('cake_logo.png', array('alt' => 'CakePHP'))?>
```

Cela affichera :

```
<img src="/img/cake_logo.png" alt="CakePHP" />
```

Pour créer une image avec un lien, spécifiez l'adresse en utilisant l'option `url dans $htmlAttributes`.

```
<?php echo $html->image("recettes/6.jpg", array(
    "alt" => "Brownies",
    'url' => array('controller' => 'recettes', 'action' => 'voir', 6)
)); ?>
```

Devrait afficher :

```
<a href="/fr/recettes/voir/6">
    <img src="/img/recettes/6.jpg" alt="Brownies" />
</a>
```

7.4.1.7 link

link(string $title, mixed $url = null, array $options = array(), string $confirmMessage = false)

Méthode universelle pour créer des liens HTML. Utilisez `$options` pour spécifier les attributs de l'élément et s'il faut ou non que `$title` soit échappé.

```
<?php echo $this->Html->link('Entrez', '/pages/home',
array('class'=>'bouton','target'=>'_blank')); ?>
```

Affichera :

```
<a href="/fr/pages/home" class="bouton" target="_blank">Entrez</a>
```

Spécifiez `$confirmMessage` pour afficher un dialogue `confirm()` en javascript.

```
<?php echo $this->Html->link(
    'Supprimer',
    array('controller'=>'recettes', 'action'=>'supprimer', 6),
    array(),
    "Ã tes-vous sÃ»r de vouloir effacer cette recette ?"
);?>
```

Affichera :

```
<a href="/fr/recettes/supprimer/6" onclick="return confirm('Ã tes-vous sÃ»r de vouloir effacer
```

Des chaînes de requête peuvent aussi être créées avec `link()`.

```
<?php echo $this->Html->link('Voir image', array(
    'controller' => 'images',
    'action' => 'voir',
    1,
    '?' => array( 'height' => 400, 'width' => 500))
);
```

Affichera :

```
<a href="/fr/images/voir/1?height=400&width=500">Voir image</a>
```

Les caractères spéciaux HTML dans `$title` seront convertis en entités HTML. Pour désactiver cette conversion, mettre l'option 'escape' à false dans le tableau `$options`.

```
<?php
echo $this->Html->link(
    $this->Html->image("recettes/6.jpg", array("alt" => "Brownies")),
    "recettes/voir/6",
    array('escape'=>false)
);
?>
```

Affichera :

```

```

```
<a href="/fr/recettes/voir/6">
    <img src="/img/recettes/6.jpg" alt="Brownies" />
</a>
```

Vous pouvez aussi vérifier la méthode HtmlHelper::url pour avoir différents exemples sur les types d'url.

7.4.1.8 tag

tag(string $tag, string $text, array $htmlAttributes, boolean $escape = false)

Retourne le texte entouré du tag spécifié. Si aucun texte n'est spécifié, seul le <tag> ouvrant sera retourné.

```
<?php echo $html->tag('span', 'Bonjour le Monde.', array('class' =>
'bienvenue'));?>

// Affichera
<span class="bienvenue">Bonjour le Monde.</span>

// Aucun texte spécifié
<?php echo $html->tag('span', null, array('class' => 'bienvenue'));?>

// Affichera
<span class="bienvenue">
```

7.4.1.9 div

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

div(string $class, string $text, array $options)

Used for creating div-wrapped sections of markup. The first parameter specifies a CSS class, and the second is used to supply the text to be wrapped by div tags. If the last parameter has been set to true, $text will be printed HTML-escaped.

```
<?php echo $this->Html->div('error', 'Please enter your credit card
number.');?>

//Output
<div class="error">Please enter your credit card number.</div>
```

If $text is set to null, only an opening div tag is returned.

```
<?php echo $this->Html->div('', null, array('id' => 'register'));?>

//Output
<div id="register" class="register">
```

7.4.1.10 para

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

para(string $class, string $text, array $htmlAttributes, boolean $escape = false)

Returns a text wrapped in a CSS-classed <p> tag. If no text is supplied, only a starting <p> tag is returned.

```
<?php echo $this->Html->para(null, 'Hello World.');?>

//Output
<p>Hello World.</p>
```

7.4.1.11 script

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

script(mixed $url, mixed $options)

Creates link(s) to a javascript file. If key `inline` is set to false in $options, the link tags are added to the $scripts_for_layout variable which you can print inside the head tag of the document.

Include a script file into the page. `$options['inline']` controls whether or not a script should be returned inline or added to $scripts_for_layout. `$options['once']` controls, whether or not you want to include this script once per request or more than once.

You can also use $options to set additional properties to the generated script tag. If an array of script tags is used, the attributes will be applied to all of the generated script tags.

This method of javascript file inclusion assumes that the javascript file specified resides inside the /app/webroot/js directory.

```
<?php echo $this->Html->script('scripts'); ?>
```

Will output:

```
<script type="text/javascript" href="/fr/js/scripts.js"></script>
```

You can link to files with absolute paths as well to link files that are not in `app/webroot/js`

```
<?php echo $this->Html->script('/otherdir/script_file'); ?>
```

The first parameter can be an array to include multiple files.

```
<?php echo $this->Html->script(array('jquery','wysiwyg','scripts')); ?>
```

Will output:

```
<script type="text/javascript" href="/fr/js/jquery.js"></script>
<script type="text/javascript" href="/fr/js/wysiwyg.js"></script>
<script type="text/javascript" href="/fr/js/scripts.js"></script>
```

7.4.1.12 scriptBlock

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`scriptBlock($code, $options = array())`

Generate a code block containing `$code` set `$options['inline']` to false to have the script block appear in `$scripts_for_layout`. Also new is the ability to add attributes to script tags. `$this->Html->scriptBlock('stuff', array('defer' => true));` will create a script tag with `defer="defer"` attribute.

7.4.1.13 scriptStart

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`scriptStart($options = array())`

Begin a buffering code block. This code block will capture all output between `scriptStart()` and `scriptEnd()` and create an script tag. Options are the same as `scriptBlock()`

7.4.1.14 scriptEnd

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`scriptEnd()`

End a buffering script block, returns the generated script element or null if the script block was opened with inline = false.

An example of using `scriptStart()` and `scriptEnd()` would be:

```
$this->Html->scriptStart(array('inline' => false));

echo $this->Js->alert('I am in the javascript');

$this->Html->scriptEnd();
```

7.4.1.15 tableHeaders

`tableHeaders(array $names, array $trOptions = null, array $thOptions = null)`

Crée une rangée de cellules d'en tête de tableau, à placer à l'intérieur des tags <table>.

```
<?php echo $html->tableHeaders(array('Date','Titre','Actif'));?> //Output
<tr><th>Date</th><th>Titre</th><th>Actif</th></tr>
```

```
<?php echo $html->tableHeaders(
    array('Date','Titre','Actif'),
    array('class' => 'status'),
    array('class' => 'product_table')
);?>

// Affichera
<tr class="status">
     <th class="product_table">Date</th>
     <th class="product_table">Titre</th>
     <th class="product_table">Actif</th>
</tr>
```

7.4.1.16 tableCells

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

tableCells(array $data, array $oddTrOptions = null, array $evenTrOptions = null, $useCount = false, $continueOddEven = true)

Creates table cells, in rows, assigning <tr> attributes differently for odd- and even-numbered rows. Wrap a single table cell within an array() for specific <td>-attributes.

```
<?php 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'),
));
?>

//Output
<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>

<?php echo $this->Html->tableCells(array(
    array('Jul 7th, 2007', array('Best Brownies', array('class'=>'highlight')) ,
'Yes'),
    array('Jun 21st, 2007', 'Smart Cookies', 'Yes'),
    array('Aug 1st, 2006', 'Anti-Java Cake', array('No', array('id'=>'special'))),
));
?>

//Output
<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>

<?php 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>
```

View more details about the tableCells function in the API

7.4.1.17 url

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`url(mixed $url = NULL, boolean $full = false)`

Returns an URL pointing to a combination of controller and action. If $url is empty, it returns the REQUEST_URI, otherwise it generates the url for the controller and action combo. If full is true, the full base URL will be prepended to the result.

```
<?php echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "view",
    "bar"));?>

// Output
/posts/view/bar
```

Here are a few more usage examples:

URL with named parameters

```
<?php echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "view",
    "foo" => "bar"));
?>

// Output
/posts/view/foo:bar
```

URL with extension

```
<?php echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "list",
    "ext" => "rss"));
?>
```

```
// Output
/posts/list.rss
```

URL (starting with '/') with the full base URL prepended.

```
<?php echo $this->Html->url('/posts', true); ?>

//Output
http://somedomain.com/posts
```

URL with GET params and named anchor

```
<?php echo $this->Html->url(array(
    "controller" => "posts",
    "action" => "search",
    "?" => array("foo" => "bar"),
    "#" => "first"));
?>

//Output
/posts/search?foo=bar#first
```

For further information check Router::url in the API.

7.4.2 Changing the tags output by HtmlHelper

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The built in tag sets for `HtmlHelper` are XHTML compliant, however if you need to generate HTML for HTML4 you will need to create and load a new tags config file containing the tags you'd like to use. To change the tags used create `app/config/tags.php` containing:

```
$tags = array(
    'metalink' => '<link href="%s"%s >',
    'input' => '<input name="%s" %s >',
    //...
);
```

You can then load this tag set by calling `$this->Html->loadConfig('tags');`

7.4.3 Creating breadcrumb trails with HtmlHelper

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

CakePHP has the built in ability to automatically create a breadcrumb trail in your app. To set this up, first add something similar to the following in your layout template.

```
    echo $this->Html->getCrumbs(' > ','Home');
```

Now, in your view you'll want to add the following to start the breadcrumb trails on each of the pages.

```
    $this->Html->addCrumb('Users', '/users');
    $this->Html->addCrumb('Add User', '/users/add');
```

This will add the output of "**Home > Users > Add User**" in your layout where getCrumbs was added.

7.5 Js

Depuis le début, le support de CakePhp pour Javascript a été fait avec Prototype/Scriptaculous. Bien que nous pensons toujours qu'elles sont d'excellentes bibliothèques Javascript, la communauté a demandé le support d'autres bibliothèques. Plutôt que d'enlever Prototype en faveur d'une autre bibliothèque Javascript, nous avons créé un assistant basé sur un adaptateur et nous avons inclus 3 des bibliothèques les plus demandées : Prototype/Scriptaculous, Mootools/Mootools-more et jQuery/jQueryUI. Et puisque l'API n'est pas aussi extensible que le précédent assistant Ajax, nous estimons que la solution basée sur un adaptateur permet une solution plus extensible, en donnant aux développeurs la puissance et la flexibilité dont ils ont besoin pour adresser les besoins spécifiques de leur application.

Les moteurs Javascript forment la colonne vertébrale du nouvel assistant Js. Un moteur Javascript traduit un élément Javascript abstrait en un code Javascript concret, spécifique de la bibliothèque Javascript qui a été utilisée. En complément, ils créent, pour les autres, un système extensible à utiliser.

7.5.1 Utilisation d'un moteur Javascript Spécifique

Avant tout, téléchargez votre bibliothèque Javascript préférée et placez la dans `app/webroot/js`

Ensuite vous devez inclure votre bibliothèque dans votre page. Pour l'inclure dans toutes les pages, ajoutez cette ligne à la section <head> de `app/views/layouts/default.ctp` (copiez ce fichier depuis `cake/libs/view/layouts/default.ctp` si vous n'avez pas créé le votre).

```
echo $this->Html->script('jquery'); // Inclue la bibliothèque JQuery
```

Remplacez `jquery` par le nom du fichier de votre bibliothèque (.js sera ajouté au nom).

Par défaut les scripts sont mis en cache, et vous devez explicitement écrire le contenu du cache. Pour ce faire, ajoutez cette ligne à la fin de chaque page juste avant la balise `</body>` tag

```
echo $this->Js->writeBuffer(); // Ecrit les scripts mis en cache
```

Vous devez inclure la bibliothèque dans votre page et écrire le contenu du cache pour que le helper fonctionne.

La sélection du moteur Javascript est déclarée quand vous incluez le helper dans votre contrôleur.

```
var $helpers = array('Js' => array('Jquery'));
```

Le code ci-dessus utiliserai le moteur JQuery dans l'instance du JsHelper dans vos vues. Si vous ne déclarez pas un moteur spécifique, le moteur JQuery sera utilisé par défaut. Comme mentioné précédemment, il y a trois moteurs implémentés dans le noyaux, mais nous encourageons la communauté à étendre la compatibilité des bibliothèques

Using jQuery with other libraries

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The jQuery library, and virtually all of its plugins are constrained within the jQuery namespace. As a general rule, "global" objects are stored inside the jQuery namespace as well, so you shouldn't get a clash between jQuery and any other library (like Prototype, MooTools, or YUI).

That said, there is one caveat: **By default, jQuery uses "$" as a shortcut for "jQuery"**

To override the "$" shortcut, use the jQueryObject variable.

```
$this->Js->JqueryEngine->jQueryObject = '$j';
print $this->Html->scriptBlock('var $j = jQuery.noConflict();',
    array('inline' => false)); //Tell jQuery to go into noconflict mode
```

7.5.1.1 Using the JsHelper inside customHelpers

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Declare the JsHelper in the `$helpers` array in your customHelper.

```
var $helpers = array('Js');
```

It is not possible to declare a javascript engine inside a custom helper. Doing that will have no effect.

If you are willing to use an other javascript engine than the default, do the helper setup in your controller as follows:

```
var $helpers = array(
    'Js' => array('Prototype'),
    'CustomHelper'
);
```

Be sure to declare the JsHelper and its engine **on top** of the `$helpers` array in your controller.

The selected javascript engine may disappear (replaced by the default) from the jsHelper object in your helper, if you miss to do so and you will get code that does not fit your javascript library.

7.5.2 Création d'un moteur Javascript

Les helpers de moteur Javascript suivent les conventions du helper normal avec quelques restrictions additionnelles. Ils doivent avoir le suffix `Engine`. `DojoHelper` n'est pas correct, `DojoEngineHelper` est correct. De plus, ils doivent étendre `JsBaseEngineHelper` afin de tirer partie au maximum de la nouvelle API.

7.5.3 Javascript engine usage

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The `JsHelper` provides a few methods, and acts as a facade for the the Engine helper. You should not directly access the Engine helper except in rare occasions. Using the facade features of the `JsHelper` allows you to leverage the buffering and method chaining features built-in; (method chaining only works in PHP5).

The `JsHelper` by default buffers almost all script code generated, allowing you to collect scripts throughout the view, elements and layout, and output it in one place. Outputting buffered scripts is done with `$this->Js->writeBuffer();` this will return the buffer contents in a script tag. You can disable buffering wholesale with the `$bufferScripts` property or setting `buffer => false` in methods taking `$options`.

Since most methods in Javascript begin with a selection of elements in the DOM, `$this->Js->get()` returns a $this, allowing you to chain the methods using the selection. Method chaining allows you to write shorter, more expressive code. It should be noted that method chaining **Will not** work in PHP4.

```
$this->Js->get('#foo')->event('click', $eventCode);
```

Is an example of method chaining. Method chaining is not possible in PHP4 and the above sample would be written like:

```
$this->Js->get('#foo');
$this->Js->event('click', $eventCode);
```

Common options

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

In attempts to simplify development where Js libraries can change, a common set of options is supported by `JsHelper`, these common options will be mapped out to the library specific options internally. If you are not planning on switching Javascript libraries, each library also supports all of its native callbacks and options.

Callback wrapping

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

By default all callback options are wrapped with the an anonymous function with the correct arguments. You can disable this behavior by supplying the `wrapCallbacks = false` in your options array.

7.5.3.1 Working with buffered scripts

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

One drawback to previous implementation of 'Ajax' type features was the scattering of script tags throughout your document, and the inability to buffer scripts added by elements in the layout. The new JsHelper if used correctly avoids both of those issues. It is recommended that you place `$this->Js->writeBuffer()` at the bottom of your layout file above the `</body>` tag. This will allow all scripts generated in layout elements to be output in one place. It should be noted that buffered scripts are handled separately from included script files.

`writeBuffer($options = array())`

Writes all Javascript generated so far to a code block or caches them to a file and returns a linked script.

Options

script block inline if `cache` is also true, a script link tag will be generated. (default true) file and linked in (default false) being cleared (default true) (default true) be wrapped in <![CDATA[...]]> (default true)

Creating a cache file with `writeBuffer()` requires that `webroot/js` be world writable and allows a browser to cache generated script resources for any page.

buffer($content)

Add `$content` to the internal script buffer.

`getBuffer($clear = true)`

Get the contents of the current buffer. Pass in false to not clear the buffer at the same time.

Buffering methods that are not normally buffered

Some methods in the helpers are buffered by default. The engines buffer the following methods by default:

- event
- sortable
- drag
- drop
- slider

Additionally you can force any other method in JsHelper to use the buffering. By appending an boolean to the end of the arguments you can force other methods to go into the buffer. For example the `each()` method does not normally buffer.

```
$this->Js->each('alert("whoa!");', true);
```

The above would force the `each()` method to use the buffer. Conversely if you want a method that does buffer to not buffer, you can pass a `false` in as the last argument.

```
$this->Js->event('click', 'alert("whoa!");', false);
```

This would force the event function which normally buffers to return its result.

7.5.4 Methods

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The core Javascript Engines provide the same feature set across all libraries, there is also a subset of common options that are translated into library specific options. This is done to provide end developers with as unified an API as possible. The following list of methods are supported by all the Engines included in the CakePHP core. Whenever you see separate lists for `Options` and `Event Options` both sets of parameters are supplied in the `$options` array for the method.

`object($data, $options = array())`

Converts values into JSON. There are a few differences between this method and JavascriptHelper::object(). Most notably there is no affordance for `stringKeys` or `q` options found in the JavascriptHelper. Furthermore `$this->Js->object();` cannot make script tags.

Options:

data. data.

Example Use:

```
$json = $this->Js->object($data);
```

`sortable($options = array())`

Sortable generates a javascript snippet to make a set of elements (usually a list) drag and drop sortable.

The normalized options are:

Options

element will start sort action. sortable into final position. before sorting starts.

Event Options

completes.

Other options are supported by each Javascript library, and you should check the documentation for your javascript library for more detailed information on its options and parameters.

Example use:

```
$this->Js->get('#my-list');
    $this->Js->sortable(array(
        'distance' => 5,
        'containment' => 'parent',
        'start' => 'onStart',
        'complete' => 'onStop',
        'sort' => 'onSort',
        'wrapCallbacks' => false
    ));
```

Assuming you were using the jQuery engine, you would get the following code in your generated Javascript block:

```
$("#myList").sortable({containment:"parent", distance:5, sort:onSort, start:onStart,
stop:onStop});
```

`request($url, $options = array())`

Generate a javascript snippet to create an `XmlHttpRequest` or 'AJAX' request.

Event Options

initialization.

Options

defaults to GET in more libraries request. request. are supported. Default is html for most libraries. should be eval'ed. be treated as a callback. Useful for supplying `$options['data']` as another Javascript expression.

Example use

```
$this->Js->event('click',
$this->Js->request(array(
'action' => 'foo', param1), array(
'async' => true,
'update' => '#element')));
```

`get($selector)`

Set the internal 'selection' to a CSS selector. The active selection is used in subsequent operations until a new selection is made.

```
$this->Js->get('#element');
```

The `JsHelper` now will reference all other element based methods on the selection of `#element`. To change the active selection, call `get()` again with a new element.

`drag($options = array())`

Make an element draggable.

Options

an array(x, y) box for the draggable element.

Event Options

release)

Example use

```
$this->Js->get('#element');
$this->Js->drag(array(
    'container' => '#content',
    'start' => 'onStart',
    'drag' => 'onDrag',
    'stop' => 'onStop',
    'snapGrid' => array(10, 10),
    'wrapCallbacks' => false
));
```

If you were using the jQuery engine the following code would be added to the buffer.

```
$("#element").draggable({containment:"#content", drag:onDrag, grid:[10,10], start:onStart,
stop:onStop});
```

`drop($options = array())`

Make an element accept draggable elements and act as a dropzone for dragged elements.

Options

accept. draggable is over.

Event Options

the drop zone. zone. drop zone without being dropped.

Example use

```
$this->Js->get('#element');
$this->Js->drop(array(
    'accept' => '.items',
    'hover' => 'onHover',
    'leave' => 'onExit',
    'drop' => 'onDrop',
    'wrapCallbacks' => false
));
```

If you were using the jQuery engine the following code would be added to the buffer:

```
<code class=
"php">$("#element").droppable({accept:".items", drop:onDrop, out:onExit,
over:onHover});</code>
```

"Note" about MootoolsEngine::drop

Droppables in Mootools function differently from other libraries. Droppables are implemented as an extension of Drag. So in addtion to making a get() selection for the droppable element. You must also provide a selector rule to the draggable element. Furthermore, Mootools droppables inherit all options from Drag.

`slider()`

Create snippet of Javascript that converts an element into a slider ui widget. See your libraries implementation for additional usage and features.

Options

sliding. 'vertical' or 'horizontal' will have.

Events

updated handle

Example use

```
$this->Js->get('#element');
$this->Js->slider(array(
    'complete' => 'onComplete',
    'change' => 'onChange',
    'min' => 0,
    'max' => 10,
    'value' => 2,
    'direction' => 'vertical',
    'wrapCallbacks' => false
));
```

If you were using the jQuery engine the following code would be added to the buffer:

```
$("#element").slider({change:onChange, max:10, min:0, orientation:"vertical",
stop:onComplete, value:2});
```

`effect($name, $options = array())`

Creates a basic effect. By default this method is not buffered and returns its result.

Supported effect names

The following effects are supported by all JsEngines

Options

Accepted values are 'slow', 'fast'. Not all effects use the speed option.

Example use

If you were using the jQuery engine.

```
$this->Js->get('#element');
$result = $this->Js->effect('fadeIn');

//$result contains $("#foo").fadeIn();
```

`event($type, $content, $options = array())`

Bind an event to the current selection. `$type` can be any of the normal DOM events or a custom event type if your library supports them. `$content` should contain the function body for the callback. Callbacks will be wrapped with `function (event) { ... }` unless disabled with the `$options`.

Options

anonymous function. (defaults to true) (defaults to true)

Example use

```
$this->Js->get('#some-link');
$this->Js->event('click', $this->Js->alert('hey you!'));
```

If you were using the jQuery library you would get the following Javascript code.

```
$('#some-link').bind('click', function (event) {
    alert('hey you!');
    return false;
});
```

You can remove the `return false;` by passing setting the `stop` option to false.

```
$this->Js->get('#some-link');
$this->Js->event('click', $this->Js->alert('hey you!'), array('stop' =>
false));
```

If you were using the jQuery library you would the following Javascript code would be added to the buffer. Note that the default browser event is not cancelled.

```
$('#some-link').bind('click', function (event) {
    alert('hey you!');
});
```

`domReady($callback)`

Creates the special 'DOM ready' event. `writeBuffer()` automatically wraps the buffered scripts in a domReady method.

each($callback)

Create a snippet that iterates over the currently selected elements, and inserts `$callback`.

Example

```
$this->Js->get('div.message');
$this->Js->each('$(this).css({color: "red"});');
```

Using the jQuery engine would create the following Javascript

```
$('div.message').each(function () { $(this).css({color: "red"});});
```

alert($message)

Create a javascript snippet containing an `alert()` snippet. By default, `alert` does not buffer, and returns the script snippet.

```
$alert = $this->Js->alert('Hey there');
```

confirm($message)

Create a javascript snippet containing a `confirm()` snippet. By default, `confirm` does not buffer, and returns the script snippet.

```
$alert = $this->Js->confirm('Are you sure?');
```

prompt($message, $default)

Create a javascript snippet containing a `prompt()` snippet. By default, `prompt` does not buffer, and returns the script snippet.

```
$prompt = $this->Js->prompt('What is your favorite color?', 'blue');
```

submit()

Create a submit input button that enables `XmlHttpRequest` submitted forms. Options can include both those for FormHelper::submit() and JsBaseEngine::request(), JsBaseEngine::event();

Forms submitting with this method, cannot send files. Files do not transfer over `XmlHttpRequest` and require an iframe, or other more specialized setups that are beyond the scope of this helper.

Options

the request. Using confirm, does not replace any `before` callback methods in the generated XmlHttpRequest. tag in addition to the link. callback wrapping.

Example use

```
echo $this->Js->submit('Save', array('update' => '#content'));
```

Will create a submit button with an attached onclick event. The click event will be buffered by default.

```
echo $this->Js->submit('Save', array('update' => '#content', 'div' => false,
'type' => 'json', 'async' => false));
```

Shows how you can combine options that both `FormHelper::submit()` and `Js::request()` when using submit.

`link($title, $url = null, $options = array())`

Create an html anchor element that has a click event bound to it. Options can include both those for HtmlHelper::link() and JsBaseEngine::request(), JsBaseEngine::event(); `$htmlAttributes` is used to specify additional options that are supposed to be appended to the generated anchor element. If an option is not part of the standard attributes or `$htmlAttributes` it will be passed to `request()` as an option. If an id is not supplied, a randomly generated one will be created for each link generated.

Options

sending the event. htmlAttributes. Standard attributes are class, id, rel, title, escape, onblur and onfocus. tag in addition to the link.

Example use

```
echo $this->Js->link('Page 2', array('page' => 2), array('update' =>
'#content'));
```

Will create a link pointing to `/page:2` and updating #content with the response.

You can use the `htmlAttributes` option to add in additional custom attributes.

```
echo $this->Js->link('Page 2', array('page' => 2), array(
    'update' =&gt; '#content',
    'htmlAttributes' =&gt; array('other' =&gt; 'value')
));


//Creates the following html
<a href="/fr/posts/index/page:2" other="value">Page 2</a>
```

`serializeForm($options = array())`

Serialize the form attached to $selector. Pass `true` for $isForm if the current selection is a form element. Converts the form or the form element attached to the current selection into a string/json object (depending on the library implementation) for use with XHR operations.

Options

input? (defaults to false) used inside another JS statement? (defaults to false)

Setting inline == false allows you to remove the trailing `;`. This is useful when you need to serialize a form element as part of another Javascript operation, or use the serialize method in an Object literal.

`redirect($url)`

Redirect the page to `$url` using `window.location`.

`value($value)`

Converts a PHP-native variable of any type to a JSON-equivalent representation. Escapes any string values into JSON compatible strings. UTF-8 characters will be escaped.

7.5.5 Ajax Pagination

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Much like Ajax Pagination in 1.2, you can use the JsHelper to handle the creation of Ajax pagination links instead of plain HTML links.

7.5.5.1 Making Ajax Links

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Before you can create ajax links you must include the Javascript library that matches the adapter you are using with `JsHelper`. By default the `JsHelper` uses jQuery. So in your layout include jQuery (or whichever library you are using). Also make sure to include `RequestHandlerComponent` in your components. Add the following to your controller:

```
var $components = array('RequestHandler');
var $helpers = array('Js');
```

Next link in the javascript library you want to use. For this example we'll be using jQuery.

```
echo $this->Html->script('jquery');
```

Similar to 1.2 you need to tell the `PaginatorHelper` that you want to make Javascript enhanced links instead of plain HTML ones. To do so you use `options()`

```
$this->Paginator->options(array(
    'update' => '#content',
    'evalScripts' => true
));
```

The `PaginatorHelper` now knows to make javascript enhanced links, and that those links should update the `#content` element. Of course this element must exist, and often times you want to wrap `$content_for_layout` with a div matching the id used for the `update` option. You also should set `evalScripts` to true if you are using the Mootools or Prototype adapters, without `evalScripts` these libraries will not be able to chain requests together. The `indicator` option is not supported by `JsHelper` and will be ignored.

You then create all the links as needed for your pagination features. Since the `JsHelper` automatically buffers all generated script content to reduce the number of `<script>` tags in your source code you **must** call write the buffer out. At the bottom of your view file. Be sure to include:

```
echo $this->Js->writeBuffer();
```

If you omit this you will **not** be able to chain ajax pagination links. When you write the buffer, it is also cleared, so you don't have worry about the same Javascript being output twice.

Adding effects and transitions

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Since `indicator` is no longer supported, you must add any indicator effects yourself.

```
<html>
    <head>
        <?php echo $this->Html->script('jquery'); ?>
        //more stuff here.
    </head>
    <body>
    <div id="content">
        <?php echo $content_for_layout; ?>
    </div>
    <?php echo $this->Html->image('indicator.gif', array('id' =>
'busy-indicator')); ?>
    </body>
</html>
```

Remember to place the indicator.gif file inside app/webroot/img folder. You may see a situation where the indicator.gif displays immediately upon the page load. You need to put in this css `#busy-indicator { display:none; }` in your main css file.

With the above layout, we've included an indicator image file, that will display a busy indicator animation that we will show and hide with the `JsHelper`. To do that we need to update our `options()` function.

```
$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' => false)),
));
```

This will show/hide the busy-indicator element before and after the `#content` div is updated. Although `indicator` has been removed, the new features offered by `JsHelper` allow for more control and more complex effects to be created.

7.6 Javascript

L'assistant Javascript est utilisé dans l'aide à la création de tags et blocs de code Javascript bien formatés. Il y a plusieurs méthodes dont certaines sont conçues pour fonctionner avec la librairie Javascript Prototype.

7.6.1 Methods

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`codeBlock($script, $options = array('allowCache'=>true,'safe'=>true,'inline'=>true), $safe)`

- string $script - The JavaScript to be wrapped in SCRIPT tags
- array $options - Set of options:
  - allowCache: boolean, designates whether this block is cacheable using the current cache settings.
  - safe: boolean, whether this block should be wrapped in CDATA tags. Defaults to helper's object configuration.
  - inline: whether the block should be printed inline, or written to cached for later output (i.e. $scripts_for_layout).
- boolean $safe - DEPRECATED. Use $options['safe'] instead

codeBlock returns a formatted script element containing $script. But can also return null if Javascript helper is set to cache events. See JavascriptHelper::cacheEvents(). And can write in `$scripts_for_layout` if you set $options['inline'] to false.

`blockEnd()`

Ends a block of cached Javascript. Can return either a end script tag, or empties the buffer, adding the contents to the cachedEvents array. Its return value depends on the cache settings. See JavascriptHelper::cacheEvents()

`link($url, $inline = true)`

- mixed $url - String URL to JavaScript file, or an array of URLs.
- boolean $inline If true, the <script> tag will be printed inline, otherwise it will be printed in `$scripts_for_layout`

Creates a javascript link to a single or many javascript files. Can output inline or in `$scripts_for_layout`.

If the filename is prefixed with "/", the path will be relative to the base path of your application. Otherwise, the path will be relative to your JavaScript path, usually webroot/js.

`escapeString($string)`

- string $script - String that needs to get escaped.

Escape a string to be JavaScript friendly. Allowing it to safely be used in javascript blocks.

The characters that are escaped are:

- "\r\n" => '\n'
- "\r" => '\n'
- "\n" => '\n'
- '"' => '\"'

- """ => "\\'"

event($object, $event, $observer, $useCapture)

- string $object - DOM Object to be observed.
- string $event - type of event to observe ie 'click', 'over'.
- string $observer - Javascript function to call when event occurs.
- array $options - Set options: useCapture, allowCache, safe
    - boolean $options['useCapture'] - Whether to fire the event in the capture or bubble phase of event handling. Defaults false
    - boolean $options['allowCache'] - See JavascriptHelper::cacheEvents()
    - boolean $options['safe'] - Indicates whether <script /> blocks should be written 'safely,' i.e. wrapped in CDATA blocks

Attach a javascript event handler specified by $event to an element DOM element specified by $object. Object does not have to be an ID reference it can refer to any valid javascript object or CSS selectors. If a CSS selector is used the event handler is cached and should be retrieved with JavascriptHelper::getCache(). This method requires the Prototype library.

cacheEvents($file, $all)

- boolean $file - If true, code will be written to a file
- boolean $all - If true, all code written with JavascriptHelper will be sent to a file

Allows you to control how the JavaScript Helper caches event code generated by event(). If $all is set to true, all code generated by the helper is cached and can be retrieved with getCache() or written to file or page output with writeCache().

getCache($clear)

- boolean $clear - If set to true the cached javascript is cleared. Defaults to true.

Gets (and clears) the current JavaScript event cache

writeEvents($inline)

- boolean $inline - If true, returns JavaScript event code. Otherwise it is added to the output of $scripts_for_layout in the layout.

Returns cached javascript code. If $file was set to true with cacheEvents(), code is cached to a file and a script link to the cached events file is returned. If inline is true, the event code is returned inline. Else it is added to the $scripts_for_layout for the page.

includeScript($script)

- string $script - File name of script to include.

Includes the named $script. If $script is left blank the helper will include every script in your app/webroot/js directory. Includes the contents of each file inline. To create a script tag with an src attribute use link().

object($data, $options)

- array $data - Data to be converted
- array $options - Set of options: block, prefix, postfix, stringKeys, quoteKeys, q
    - boolean $options['block'] - Wraps return value in a <script /> block if true. Defaults to false.

- string $options['prefix'] - Prepends the string to the returned data.
- string $options['postfix'] - Appends the string to the returned data.
- array $options['stringKeys'] - A list of array keys to be treated as a string.
- boolean $options['quoteKeys'] - If false, treats $stringKey as a list of keys *not* to be quoted. Defaults to true.
- string $options['q'] - The type of quote to use.

Generates a JavaScript object in JavaScript Object Notation (JSON) from $data array.

7.7 Number

L'assistant Number contient des méthodes pratiques qui permettent l'affichage des nombres dans des formats courants au sein de vos vues. Ces méthodes incluent différentes manières de formater les monnaies, les pourcentages, les tailles de données, de formater les nombres pour des besoins spécifiques et aussi pour vous donner plus de flexibilité en formatant les nombres.

Toutes ces fonctions renvoie le nombre formaté. Elles n'affichent pas automatiquement le résultat dans les vues.

7.7.1 currency

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`currency(mixed $number, string $currency= 'USD', $options = array())`

This method is used to display a number in common currency formats (EUR,GBP,USD). Usage in a view looks like:

```
<?php echo $this->Number->currency($number,$currency); ?>
```

The first parameter, $number, should be a floating point number that represents the amount of money you are expressing. The second parameter is used to choose a predefined currency formatting scheme:

$currency	1234.56, formatted by currency type
EUR	€ 1.236,33
GBP	£ 1,236.33
USD	$ 1,236.33

The third parameter is an array of options for further defining the output. The following options are available:

Option	Description
before	The currency symbol to place before whole numbers ie. '$'
after	The currency symbol to place after decimal numbers ie. 'c'. Set to boolean false to use no decimal symbol. eg. 0.35 => $0.35.

zero	The text to use for zero values, can be a string or a number. ie. 0, 'Free!'
places	Number of decimal places to use. ie. 2
thousands	Thousands separator ie. ','
decimals	Decimal separator symbol ie. '.'
negative	Symbol for negative numbers. If equal to '()', the number will be wrapped with (and)
escape	Should the output be htmlentity escaped? Defaults to true

If a non-recognized $currency value is supplied, it is prepended to a USD formatted number. For example:

```
<?php echo $this->Number->currency('1234.56', 'FOO'); ?>

//Outputs:
FOO 1,234.56
```

7.7.2 precision

precision (mixed $nombre, int $precision = 3)

Cette méthode affiche un nombre avec le degré de précision spécifié (nombre de décimales). Il sera arrondi pour conserver ce niveau de précision.

```
<?php echo $number->precision(456.91873645, 2 ); ?>

// Affiche :
456.92
```

7.7.3 toPercentage

toPercentage(mixed $nombre, int $precision = 2)

Comme precision(), cette méthode formate un nombre en fonction de la précision fournie (les nombres sont arrondis pour répondre à la précision donnée). Cette méthode exprime également le nombre comme un pourcentage et complète l'affichage avec le signe "pour cent".

```
<?php echo $number->toPercentage(45.691873645); ?>

// Affiche :
45.69%
```

7.7.4 toReadableSize

toReadableSize(string $taille_donnee)

Cette méthode formate des tailles de données sous une forme compréhensible par les humains. Elle fournit un raccourci pour convertir des octets en Ko, Mo, Go et To. La taille est affichée avec un niveau de précision à deux chiffres, adapté à la taille des données fournies (par exemple : les tailles les plus grandes sont exprimées avec les unités les plus grandes) :

```
echo $number->toReadableSize(0);  // 0 octet
echo $number->toReadableSize(1024); // 1 Ko
echo $number->toReadableSize(1321205.76); // 1.26 Mo
echo $number->toReadableSize(5368709120); // 5.00 Go
```

7.7.5 format

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

format (mixed $number, mixed $options=false)

This method gives you much more control over the formatting of numbers for use in your views (and is used as the main method by most of the other NumberHelper methods). Using this method might looks like:

```
$this->Number->format($number, $options);
```

The $number parameter is the number that you are planning on formatting for output. With no $options supplied, the number 1236.334 would output as 1,236. Note that the default precision is zero decimal places.

The $options parameter is where the real magic for this method resides.

- If you pass an integer then this becomes the amount of precision or places for the function.
- If you pass an associated array, you can use the following keys:
    - ♦ places (integer): the amount of desired precision
    - ♦ before (string): to be put before the outputted number
    - ♦ escape (boolean): if you want the value in before to be escaped
    - ♦ decimals (string): used to delimit the decimal places in a number
    - ♦ thousands (string): used to mark off thousand, millions, ... places

```
echo $this->Number->format('123456.7890', array(
    'places' => 2,
    'before' => '¥ ',
    'escape' => false,
    'decimals' => '.',
    'thousands' => ','
));
// output '¥ 123,456.79'
```

7.8 Paginator

L'assistant de pagination est utilisé pour contrôler les outils de pagination comme le nombre de page ainsi que les liens précédent/suivant.

Voir aussi Tâches courantes avec cakePHP - Pagination pour plus d'informations.

7.8.1 Methods

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

options($options = array())

- mixed options() Default options for pagination links. If a string is supplied - it is used as the DOM id element to update. See #options for list of keys.

options() sets all the options for the Paginator Helper. Supported options are:

format

Format of the counter. Supported formats are 'range' and 'pages' and custom which is the default. In the default mode the supplied string is parsed and tokens are replaced with actual values. The available tokens are:

- %page% - the current page displayed.
- %pages% - total number of pages.
- %current% - current number of records being shown.
- %count% - the total number of records in the result set.
- %start% - number of the first record being displayed.
- %end% - number of the last record being displayed.

Now that you know the available tokens you can use the counter() method to display all sorts of information about the returned results, for example:

```
echo $this->Paginator->counter(array(
        'format' => 'Page %page% of %pages%,
                    showing %current% records out of %count% total,
                    starting on record %start%, ending on %end%'
));
```

separator

The separator between the actual page and the number of pages. Defaults to ' of '. This is used in conjunction with format = 'pages'

url

The url of the paginating action. url has a few sub options as well

- sort - the key that the records are sorted by
- direction - The direction of the sorting. Defaults to 'ASC'
- page - The page number to display

model

The name of the model being paginated.

escape

Defines if the title field for links should be HTML escaped. Defaults to true.

update

The DOM id of the element to update with the results of AJAX pagination calls. If not specified, regular links will be created.

indicator

DOM id of the element that will be shown as a loading or working indicator while doing AJAX requests.

`link($title, $url = array(), $options = array())`

- string $title - Title for the link.
- mixed $url Url for the action. See Router::url()
- array $options Options for the link. See options() for list of keys.

Creates a regular or AJAX link with pagination parameters

```
echo $this->Paginator->link('Sort by title on page 5',
        array('sort' => 'title', 'page' => 5, 'direction' => 'desc'));
```

If created in the view for `/posts/index` Would create a link pointing at '/posts/index/page:5/sort:title/direction:desc'

7.9 RSS

Cet exemple suppose que vous ayez un Contrôleur Posts et un modèle Post déjà créés et que vous vouliez créer une vue alternative pour les flux RSS.

Créer une version xml/rss de l'index de vos posts est vraiment simple avec CakePHP 1.2. 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 contrôleur pour y ajouter le code "rss-spécifique".

7.9.1.1 Le code du Contrôleur

C'est une bonne idée d'ajouter RequestHandler au tableau $components de votre contrôleur Posts. Cela permettra à beaucoup d'automagie de se produire.

```
var $components = array('RequestHandler');
```

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 contrôleur 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 contrôleur pourrait rester le même.

```
// Modifie l'action du Contrôleur 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 DESC'));
        $this->set(compact('posts'));
    } else {
        // 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.

7.9.1.1.1 Layout

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

An Rss layout is very simple, put the following contents in **app/views/layouts/rss/default.ctp**:

```
echo $this->Rss->header();
if (!isset($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);
```

It doesn't look like much but thanks to the power in the RssHelper its doing a lot of lifting for us. We haven't set $documentData or $channelData in the controller, however in CakePHP 1.3 your views can pass variables back to the layout. Which is where our $channelData array will come from setting all of the meta data for our feed.

Next up is view file for my posts/index. Much like the layout file we created, we need to create a

views/posts/rss/ directory and create a new index.ctp inside that folder. The contents of the file are below.

7.9.1.1.2 View

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Our view, located at `app/views/posts/rss/index.ctp`, begins by setting the $documentData and $channelData variables for the layout, these contain all the metadata for our RSS feed. This is done by using the View::set() method which is analogous to the Controller::set() method. Here though we are passing the channel's metadata back to the layout.

```
    $this->set('documentData', array(
        'xmlns:dc' => 'http://purl.org/dc/elements/1.1/'));

    $this->set('channelData', array(
        'title' => __("Most Recent Posts", true),
        'link' => $this->Html->url('/', true),
        'description' => __("Most recent posts.", true),
        'language' => 'en-us'));
```

The second part of the view generates the elements for the actual records of the feed. This is accomplished by looping through the data that has been passed to the view ($items) and using the RssHelper::item() method. The other method you can use, RssHelper::items() which takes a callback and an array of items for the feed. (The method I have seen used for the callback has always been called transformRss(). There is one downfall to this method, which is that you cannot use any of the other helper classes to prepare your data inside the callback method because the scope inside the method does not include anything that is not passed inside, thus not giving access to the TimeHelper or any other helper that you may need. The RssHelper::item() transforms the associative array into an element for each key value pair.

You will need to modify the $postLink variable as appropriate to your application.

```
    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']);
        // You should import Sanitize
        App::import('Sanitize');
        // This is the part where we clean the body text for output as the description
        // of the rss item, this needs to have only text to make sure the feed validates
        $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,
            'dc:creator' => $post['Post']['author'],
            'pubDate' => $post['Post']['created']));
    }
```

You can see above that we can use the loop to prepare the data to be transformed into XML elements. It is important to filter out any non-plain text characters out of the description, especially if you are using a rich text editor for the body of your blog. In the code above we use the TextHelper::stripLinks() method and a few methods from the Sanitize class, but we recommend writing a comprehensive text cleaning helper to really scrub the text clean. Once we have set up the data for the feed, we can then use the RssHelper::item() method to create the XML in RSS format. Once you have all this setup, you can test your RSS feed by going to your site /posts/index.rss and you will see your new feed. It is always important that you validate your RSS feed before making it live. This can be done by visiting sites that validate the XML such as Feed Validator or the w3c site at http://validator.w3.org/feed/.

You may need to set the value of 'debug' in your core configuration to 1 or to 0 to get a valid feed, because of the various debug information added automagically under higher debug settings that break XML syntax or feed validation rules.

7.10 Session

Équivalent du composant Session, l'assistant Session offre la majorité des fonctionnalités du composant et les rend disponible dans votre vue. L'assistant session est automatiquement ajouté à la vue, il n'est pas nécessaire de l'ajouter à la variable tableau `$helpers` dans le contrôleur.

La grande différence entre le composant Session et l'assistant Session est que ce dernier ne peut *pas* écrire dans la session.

Comme pour le composant Session, les données sont écrites et lues en utilisant des tableaux, comme ci-dessous :

```
    array('Utilisateur' =>
            array('pseudo' => 'super@exemple.com')
    );
```

Étant donné ce tableau, le noeud sera accessible par Utilisateur.pseudo, le point indiquant le tableau imbriqué. Cette notation est utilisée pour toutes les méthodes de l'assistant Session où une variable $key est utilisée.

7.10.1 Methods

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

read($key)	Read from the Session. Returns a string or array depending on the contents of the session.
id()	Returns the current session ID.
check($key)	Check to see if a key is in the Session. Returns a boolean on the key's existence.
flash($key)	This will return the contents of the $_SESSION.Message. It is used in conjunction with the Session Component's setFlash() method.

error()	Returns the last error in the session if one exists.

7.10.2 flash

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The flash method uses the default key set by `setFlash()`. You can also retrieve specific keys in the session. For example, the Auth component sets all of its Session messages under the 'auth' key

```
// Controller code
$this->Session->setFlash('My Message');

// In view
echo $this->Session->flash();
// outputs "<div id='flashMessage' class='message'>My Message</div>"

// output the AuthComponent Session message, if set.
echo $this->Session->flash('auth');
```

Using Flash for Success and Failure

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

In some web sites, particularly administration backoffice web applications it is often expected that the result of an operation requested by the user has associated feedback as to whether the operation succeeded or not. This is a classic usage for the flash mechanism since we only want to show the user the result once and not keep the message.

One way to achieve this is to use Session->flash() with the layout parameter. With the layout parameter we can be in control of the resultant html for the message.

In the controller you might typically have code:

```
if ($user_was_deleted) {
    $this->Session->setFlash('The user was deleted successfully.', 'flash_success');
} else {
    $this->Session->setFlash('The user could not be deleted.', 'flash_failure');
}
```

The flash_success and flash_failure parameter represents an element file to place in the root app/views/elements folder, e.g. app/views/elements/flash_success.ctp, app/views/elements/flash_failure.ctp

Inside the flash_success element file would be something like this:

```
<div class="flash flash_success">
    <?php echo $message ?>
</div>
```

The final step is in your main view file where the result is to be displayed to add simply

```
<?php echo $this->Session->flash(); ?>
```

And of course you can then add to your CSS a selector for div.flash, div.flash_success and div.flash_failure

7.11 Text

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

autoLinkEmails(string $text, array $htmlOptions=array())

Adds links to the well-formed email addresses in $text, according to any options defined in `$htmlOptions` (see HtmlHelper::link()).

```
$my_text = 'For more information regarding our world-famous pastries and desserts, contact
info@example.com';
$linked_text = $this->Text->autoLinkEmails($my_text);
```

Output:

```
For more information regarding our world-famous pastries and desserts,
contact <a href="mailto:info@example.com">info@example.com</a>
```

autoLinkUrls

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

autoLinkUrls(string $text, array $htmlOptions=array())

Same as in `autoLinkEmails()`, only this method searches for strings that start with https, http, ftp, or nntp and links them appropriately.

autoLink

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

autoLink(string $text, array $htmlOptions=array())

Performs the functionality in both `autoLinkUrls()` and `autoLinkEmails()` on the supplied `$text`. All URLs and emails are linked appropriately given the supplied `$htmlOptions`.

excerpt

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

excerpt(string $haystack, string $needle, int $radius=100, string $ending="...")

Extracts an excerpt from `$haystack` surrounding the `$needle` with a number of characters on each side determined by `$radius`, and suffixed with `$ending`. This method is especially handy for search results. The query string or keywords can be shown within the resulting document.

```
    echo $this->Text->excerpt($last_paragraph, 'method', 50);
```

Output:

```
mined by $radius, and suffixed with $ending. This method is especially handy for
search results. The query...
```

highlight

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`highlight(string $haystack, string $needle, array $options = array() )`

Highlights `$needle` in `$haystack` using the `$options['format']` string specified or a default string.

Options

- 'format' - string The piece of html with that the phrase will be highlighted
- 'html' - bool If true, will ignore any HTML tags, ensuring that only the correct text is highlighted

```
    echo $this->Text->highlight($last_sentence, 'using',array('format'=>'<span
class="highlight">\1</span>');
```

Output:

```
Highlights $needle in $haystack <span class="highlight">using</span>
the $options['format'] string specified  or a default string.
```

stripLinks

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`stripLinks($text)`

Strips the supplied `$text` of any HTML links.

toList

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`toList(array $list, $and='and', $separator=', ')`

Creates a comma-separated list where the last two items are joined with 'and'.

```
    echo $this->Text->toList($colors);
```

Output:

```
red, orange, yellow, green, blue, indigo and violet
```

truncate

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`truncate(string $text, int $length=100, array $options)`

Cuts a string to the `$length` and adds a suffix with `'ending'` if the text is longer than `$length`. If `'exact'` is passed as `false`, the truncation will occur after the next word ending. If `'html'` is passed as `true`, html tags will be respected and will not be cut off.

`$options` is used to pass all extra parameters, and has the following possible keys by default, all of which are optional:

```
array(
    'ending' => '...',
    'exact' => true,
    'html' => false
)
```

```
echo $this->Text->truncate(
    'The killer crept forward and tripped on the rug.',
    22,
    array(
        'ending' => '...',
        'exact' => false
    )
);
```

Output:

```
The killer crept...
```

trim

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`trim()`

An alias for truncate.

7.12 Time

Le Helper Time vous permet, comme il l'indique: gagnez du temps. Il permet le traitement rapide des informations se rapportant au temps. Le Helper Time a deux principales tâches qu'il peut accomplir:

1. Il peut formater les chaines de temps.
2. Il peut tester le temps (mais ne peut pas le courber, désolé).

7.12.1 Formatage

```
fromString( $date_string )
```

fromString prend une chaîne de caractères et utilise strtotime pour la convertir en date. Si la chaîne de caractères passée est un nombre alors il la convertira en un entier étant le nombre de secondes depuis l'époque d'UNIX (1er Janvier 1970 à 00:00:00 GMT). Le passage dans une chaîne de caractères de "31122010" créera un résultat non désiré étant donné qu'il le convertira en nombre de secondes depuis l'époque d'UNIX ce qui dans ce cas donnera "Dimanche 27 Décembre 1970 à 06:00:10"

```
toQuarter( $date_string, $range = false )
```

toQuarter retournera 1, 2, 3 ou 4 en fonction du trimestre dans lequel tombe la date passée en paramètre. Si range est mis à 'true', un tableau de deux éléments sera retourné avec la date de début et de fin du trimestre au format "AAAA-MM-JJ".

```
toUnix( $date_string )
```

toUnix est un conteneur, un emballage, pour fromString.

```
toAtom( $date_string )
```

toAtom retourne une chaîne de caractères date au format Atom "2008-01-12T00:00:00Z"

```
toRSS( $date_string )
```

toRSS retourne une chaîne de caractères au format RSS "Sat, 12 Jan 2008 00:00:00 -0500"

```
nice( $date_string = null )
```

nice prend une chaîne de caractères date et la retourne au format "Tue, Jan 1st 2008, 19:25".

```
niceShort( $date_string = null )
```

niceShort prend une chaîne de caractères date et la retourne au format "Jan 1st 2008, 19:25". Si l'objet date est aujourd'hui, le format sera "Today, 19:25". Si l'objet date est hier, le format sera "Yesterday, 19:25".

```
daysAsSql( $begin, $end, $fieldName, $userOffset = NULL )
```

daysAsSql retourne une chaîne de caractères au format "($field_name >= '2008-01-21 00:00:00') AND ($field_name <= '2008-01-25 23:59:59')". C'est utile si vous avez besoin de chercher les enregistrements entre deux dates inclusivement.

```
dayAsSql( $date_string, $field_name )
```

dayAsSql crée une chaîne de caractères au même format que daysAsSql mais ne nécessite qu'un seul objet date.

```
timeAgoInWords( $datetime_string, $options = array(), $backwards = null )
```

timeAgoInWords prendra une chaîne de caractères datetime (tout ce qui est interprétable par la fonction PHP strtotime() ou par le format datetime de MySQL) et la convertira en phrase conviviale tel que "3 weeks, 3 days ago". Mettre $backwards à "true" va spécialement déclarer que le temps est mis au futur ce qui utilisera le format "on 31/12/08".

Option	Description
format	un format de date; par défaut "on 31/12/08"
end	détermine le point de coupure à partir duquel il n'est plus utile d'employer des mots et où il utilisera le format de date à la place; par défaut "+1 month"

```
relativeTime( $date_string, $format = 'j/n/y' )
```

relativeTime est essentiellement un alias pour timeAgoInWords.

```
gmt( $date_string = null )
```

gmt retournera la date comme un entier mis au Greenwich Mean Time (GMT).

```
format( $format = 'd-m-Y', $date_string)
```

format est un conteneur, un emballage, pour la fonction PHP date.

Format	Sample Output
nice	Tue, Jan 1st 2008, 19:25
niceShort	Jan 1st 2008, 19:25 Today, 19:25 Yesterday, 19:25
daysAsSql	($field_name >= '2008-01-21 00:00:00') AND ($field_name <= '2008-01-25 23:59:59')
dayAsSql	($field_name >= '2008-01-21 00:00:00') AND ($field_name <= '2008-01-21 23:59:59')
timeAgoInWords relativeTime	on 21/01/08 3 months, 3 weeks, 2 days ago 7 minutes ago 2 seconds ago
gmt	1200787200

7.12.2 Testing Time

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

All of the above functions return true or false when passed a date string. `wasWithinLast` takes an additional `$time_interval` option:

```
$this->Time->wasWithinLast( $time_interval, $date_string )
```

`wasWithinLast` takes a time interval which is a string in the format "3 months" and accepts a time interval of seconds, minutes, hours, days, weeks, months and years (plural and not). If a time interval is not recognized

(for example, if it is mistyped) then it will default to days.

7.13 XML

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`serialize($data, $options = array())`

- mixed $data - The content to be converted to XML
- mixed $options - The data formatting options. For a list of valid options, see `Xml::__construct()`
  - string $options['root'] - The name of the root element, defaults to '#document'
  - string $options['version'] - The XML version, defaults to '1.0'
  - string $options['encoding'] - Document encoding, defaults to 'UTF-8'
  - array $options['namespaces'] - An array of namespaces (as strings) used in this document
  - string $options['format'] - Specifies the format this document converts to when parsed or rendered out as text, either 'attributes' or 'tags', defaults to 'attributes'
  - array $options['tags'] - An array specifying any tag-specific formatting options, indexed by tag name. See XmlNode::normalize()
  - boolean $options['slug'] - A boolean to indicate whether or not you want the string version of the XML document to have its tags run through Inflector::slug() (Which makes 'TAGNAME' became 't_a_g_n_a_m_e'). Defaults to true.

The serialize method takes an array and creates an XML string of the data. This is commonly used for serializing model data.

```
<?php
echo $this->Xml->serialize($data);
 // format will be similar to:
 // <model_name id="1" field_name="content" />
?>
```

The serialize method acts as a shortcut to instantiating the XML built-in class and using the toString method of that. If you need more control over serialization, you may wish to invoke the XML class directly.

You can modify how a data is serialized by using the *format* attribute. By default the data will be serialized as attributes. If you set the *format* as "tags" the data will be serialized as tags.

```
pr($data);
```

```
Array
(
    [Baker] => Array
        (
            [0] => Array
                (
                    [name] => The Baker
                    [weight] => heavy
                )
            [1] => Array
                (
```

```
                    [name] => The Cook
                    [weight] => light-weight
                )
        )
)
```

```
pr($this->Xml->serialize($data));
```

```
<baker>
     <baker name="The Baker" weight="heavy" />
     <baker name="The Cook" weight="light-weight" />
</baker>
```

```
pr($this->Xml->serialize($data, array('format' => 'tags')));
```

```
<baker>
    <baker>
        <name><![CDATA[The Baker]]></name>
        <weight><![CDATA[heavy]]></weight>
    </baker>
    <baker>
        <name><![CDATA[The Cook]]></name>
        <weight><![CDATA[light-weight]]></weight>
    </baker>
</baker>
```

7.13.2 elem

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The elem method allows you to build an XML node string with attributes and internal content, as well.

`string elem (string $name, $attrib = array(), mixed $content = null, $endTag = true)`

```
echo $this->Xml->elem('count', array('namespace' => 'myNameSpace'), 'content');
// generates: <myNameSpace:count>content</count>
```

If you want to wrap your text node with CDATA, the third argument should be an array containing two keys: 'cdata' and 'value'

```
echo $this->Xml->elem('count', null, array('cdata'=>true,'value'=>'content'));
// generates: <count><![CDATA[content]]></count>
```

7.13.3 header

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The `header()` method is used to output the XML declaration.

```
<?php
echo $this->Xml->header();
// generates: <?xml version="1.0" encoding="UTF-8" ?>
?>
```

You can pass in a different version number and encoding type as parameters of the header method.

```
<?php
echo $this->Xml->header(array('version'=>'1.1'));
// generates: <?xml version="1.1" encoding="UTF-8" ?>
?>
```

8 Librairies utilitaires intégrées

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

App is a very small utility library. It only contains the **import** method. But, with the import method, you can accomplish a lot.

```
// examples
App::Import('Core','File');
App::Import('Model','Post');
App::import('Vendor', 'geshi');
App::import('Vendor', 'flickr/flickr');
App::import('Vendor', 'SomeName', array('file' => 'some.name.php'));
App::import('Vendor', 'WellNamed', array('file' => 'services'.DS.'well.named.php'));
```

You can read more about it in the book or the API documentation

8.2 Inflector

	Entrée	Sortie
pluralize	Pomme, Orange, Cheval, Homme	Pommes, Oranges, Chevaux, Hommes
singularize	Pommes, Oranges, Chevaux, Hommes	pomme, Orange, Cheval, Homme
camelize	Tarte_aux_pommes, quelque_chose, chevaux_cheval	tarteAuxPommes, quelqueChose, chevauxCheval
underscore	Il est à noter que la méthode underscore convertira uniquement les mots formatés en camelCase. Les mots qui contiennent des espaces seront mis en minuscules, mais ils n'auront pas d'underscore.	
tarteAuxPommes, quelqueChose	tarte_aux_pommes, quelque_chose	
humanize	tarte_aux_pommes, quelque_chose, chevaux_cheval	Tarte aux pommes, Quelque chose, chevaux cheval
tableize	Pomme, UtilisateurProfilOption, Cheval	pommes, utilisateur_profil_options, chevaux
classify	pommes, utilisateur_profil_options, chevaux	Pomme, UtilisateurProfilOption, Cheval
variable	pommes, utilisateur_resultat, chevaux_chevaux	pommes, utilisateurResultat, chevauxChevaux
slug	Slug convertit les caractères spéciaux dans leur version latine et les caractères inconnus, ainsi que les espaces en underscores. La méthode slug attend un encodage UTF-8.	
purée de pomme	puree_de_pomme	

8.3 String

La méthode uuid est utilisée pour générer un identifiant unique suivant la RFC 4122. Une uuid est une chaîne de caractère de 128 bits dans un format de type 485fc381-e790-47a3-9794-1337c0a8fe68.

```
String::uuid(); // 485fc381-e790-47a3-9794-1337c0a8fe68
```

8.3.2 tokenize

`string tokenize ($data, $separator = ',', $leftBound = '(', $rightBound = ')')`

Segmente une chaîne en utilisant `$separator`, ignore les occurrences de `$separator` s'ils apparaissent entre `$leftBound` et `$rightBound`.

8.3.3 insert

`string insert ($string, $data, $options = array())`
La méthode insert est utilisée pour créer des gabarits de chaîne et pour permettre des remplacements clé/valeur.

```
String::insert('Mon nom est :nom et j\'ai :age ans.', array('nom' => 'Bob', 'age' =>
'65'));
// génère : "Mon nom est Bob et j'ai 65 ans."
```

8.3.4 cleanInsert

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`string cleanInsert ($string, $options = array())`
Cleans up a Set::insert formatted string with given $options depending on the 'clean' key in $options. The default method used is text but html is also available. The goal of this function is to replace all whitespace and unneeded markup around placeholders that did not get replaced by Set::insert.

You can use the following options in the options array:

```
$options = array(
    'clean' => array(
        'method' => 'text', // or html
    ),

    'before' => '',
    'after' => ''
);
```

8.4 Xml

La classe Xml fournit une façon simple pour parcourir et générer des fragments et documents XML. C'est une solution entièrement PHP qui nécessite seulement que l'extension Xml/Expat soit installée.

8.4.1 Analyse (parsing) Xml

Parser du Xml avec la classe Xml vous oblige à avoir une chaîne contenant le xml que vous voulez parser.

```
$input = '<' . '?xml version="1.0" encoding="UTF-8" ?' . '>
    <containeur>
        <element id="premier-el">
            <nom>Mon élément</name>
            <taille>20</taille>
        </element>
        <element>
            <nom>Ton élément</nom>
            <taille>30</taille>
        </element>
    </containeur>';
$xml = new Xml($input);
```

Ceci créera un objet document Xml qui pourra ensuite être manipulé, parcouru et reconverti en chaîne de caractères.

Avec l'exemple ci-dessus vous pouvez faire ce qui suit.

```
echo $xml->children[0]->children[0]->name;
// affiche 'element'

echo $xml->children[0]->children[0]->children[0]->children[0]->value;
// affiche 'Mon élément'

echo $xml->children[0]->child('element')->attributes['id'];
// affiche 'premier-el'
```

De plus, il est parfois plus facile d'obtenir des données depuis XML, si vous convertissez l'objet document XML en un tableau.

```
$xml = new Xml($input);
// Ceci convertit l'objet document XML en un tableau formaté
$xmlAsArray = Set::reverse($xml);
// Vous pouvez aussi simplement le convertir en appelant toArray();
$xmlAsArray = $xml->toArray();
```

8.5 Set

La gestion des tableaux, si elle est bien faite, peut être un outil utile et puissant pour développer un code plus intelligent et optimisé. CakePHP offre un ensemble très utile de fonctionnalités statiques dans la classe Set, qui vous permettra de faire cela.

La classe Set de CakePHP peut être appelée depuis n'importe quel modèle ou contrôleur, de la même manière qu'on appelle *Inflector*. Exemple : Set::combine().

8.5.1 Set-compatible Path syntax

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

The Path syntax is used by (for example) sort, and is used to define a path.

Usage example (using Set::sort()):

```
$a = array(
    0 => array('Person' => array('name' => 'Jeff')),
    1 => array('Shirt' => array('color' => 'black'))
);
$result = Set::sort($a, '{n}.Person.name', 'asc');
/* $result now looks like:
    Array
    (
        [0] => Array
            (
                [Shirt] => Array
                    (
                        [color] => black
                    )
            )
        [1] => Array
            (
                [Person] => Array
                    (
                        [name] => Jeff
                    )
            )
    )
*/
```

As you can see in the example above, some things are wrapped in {}'s, others not. In the table below, you can see which options are available.

Expression	Definition
{n}	Represents a numeric key
{s}	Represents a string
Foo	Any string (without enclosing brackets) is treated like a string literal.
{[a-z]+}	Any string enclosed in brackets (besides {n} and {s}) is interpreted as a regular expression.

This section needs to be expanded.

8.5.2 insert

`array Set::insert ($list, $path, $data = null)`

Insère $data dans un tableau défini par $path.

```
$a = array(
    'pages' => array('nom' => 'page')
);
$resultat = Set::insert($a, 'fichiers', array('nom' => 'fichiers'));
/* $resultat ressemble maintenant à :
    Array
```

```
    (
        [pages] => Array
            (
                [nom] => page
            )
        [fichiers] => Array
            (
                [nom] => fichiers
            )
    )
*/

$a = array(
    'pages' => array('nom' => 'page')
);
$resultat = Set::insert($a, 'pages.nom', array());
/* $resultat ressemble maintenant à :
    Array
    (
        [pages] => Array
            (
                [nom] => Array
                    (
                    )
            )
    )
*/

$a = array(
    'pages' => array(
        0 => array('nom' => 'principale'),
        1 => array('nom' => 'à propos')
    )
);
$resultat = Set::insert($a, 'pages.1.variables', array('titre' => 'titre page'));
/* $resultat ressemble maintenant à :
    Array
    (
        [pages] => Array
            (
                [0] => Array
                    (
                        [nom] => principale
                    )
                [1] => Array
                    (
                        [nom] => à propos
                        [variables] => Array
                            (
                                [titre] => titre page
                            )
                    )
            )
    )
*/
```

8.5.3 sort

`array Set::sort ($data, $path, $dir)`

Trie un tableau par une valeur quelconque, déterminée par un chemin "Set-compatible".

```
$a = array(
    0 => array('Personne' => array('nom' => 'Jeff')),
    1 => array('Chemise' => array('couleur' => 'noir'))
);
$resultat = Set::sort($a, '{n}.Personne.nom', 'asc');
/* $resultat ressemble maintenant à :
    Array
    (
        [0] => Array
            (
                [Chemise] => Array
                    (
                        [couleur] => noir
                    )
            )
        [1] => Array
            (
                [Personne] => Array
                    (
                        [nom] => Jeff
                    )
            )
    )
*/

$resultat = Set::sort($a, '{n}.Chemise', 'asc');
/* $resultat ressemble maintenant à :
    Array
    (
        [0] => Array
            (
                [Personne] => Array
                    (
                        [nom] => Jeff
                    )
            )
        [1] => Array
            (
                [Chemise] => Array
                    (
                        [couleur] => noir
                    )
            )
    )
*/

$resultat = Set::sort($a, '{n}', 'desc');
/* $resultat ressemble maintenant à :
    Array
    (
        [0] => Array
            (
                [Chemise] => Array
                    (
                        [couleur] => noir
                    )
            )
        [1] => Array
            (
                [Personne] => Array
                    (
                        [nom] => Jeff
                    )
            )
    )
```

```
*/

$a = array(
    array(7,6,4),
    array(3,4,5),
    array(3,2,1),
);

$resultat = Set::sort($a, '{n}.{n}', 'asc');
/* $resultat ressemble maintenant à :
    Array
    (
        [0] => Array
            (
                [0] => 3
                [1] => 2
                [2] => 1
            )
        [1] => Array
            (
                [0] => 3
                [1] => 4
                [2] => 5
            )
        [2] => Array
            (
                [0] => 7
                [1] => 6
                [2] => 4
            )
    )
*/
```

8.5.4 reverse

`array Set::reverse ($object)`

Set::reverse est l'opposé de set::map. Il convertit un objet en un tableau. Si $object n'est pas un objet, la fonction retournera simplement $object.

```
$result = Set::reverse(null);
// Null
$result = Set::reverse(false);
// false
$a = array(
    'Post' => array('id'=> 1, 'titre' => 'Premier Post'),
    'Commentaire' => array(
        array('id'=> 1, 'titre' => 'Premier Commentaire'),
        array('id'=> 2, 'titre' => 'Second Commentaire')
    ),
    'Tag' => array(
        array('id'=> 1, 'titre' => 'Premier Tag'),
        array('id'=> 2, 'titre' => 'Second Tag')
    ),
);
$map = Set::map($a); // Transforme $a en classe Object
/* $map ressemble maintenant à :
    stdClass Object
    (
        [_name_] => Post
        [id] => 1
```

```
        [titre] => Premier Post
        [Commentaire] => Array
            (
                [0] => stdClass Object
                    (
                        [id] => 1
                        [titre] => Premier Commentaire
                    )
                [1] => stdClass Object
                    (
                        [id] => 2
                        [titre] => Second Commentaire
                    )
            )
        [Tag] => Array
            (
                [0] => stdClass Object
                    (
                        [id] => 1
                        [titre] => Premier Tag
                    )
                [1] => stdClass Object
                    (
                        [id] => 2
                        [titre] => Second Tag
                    )
            )
    )
*/

$resultat = Set::reverse($map);
/* $resultat ressemble maintenant à :
    Array
    (
        [Post] => Array
            (
                [id] => 1
                [titre] => Premier Post
                [Commentaire] => Array
                    (
                        [0] => Array
                            (
                                [id] => 1
                                [titre] => Premier Commentaire
                            )
                        [1] => Array
                            (
                                [id] => 2
                                [titre] => Second Commentaire
                            )
                    )
                [Tag] => Array
                    (
                        [0] => Array
                            (
                                [id] => 1
                                [titre] => Premier Tag
                            )
                        [1] => Array
                            (
                                [id] => 2
                                [titre] => Second Tag
                            )
                    )
            )
```

```
    )
*/

$resultat = Set::reverse($a['Post']); // Retourne simplement le tableau
/* $resultat ressemble maintenant à :
    Array
    (
        [id] => 1
        [titre] => Premier Post
    )
*/
```

8.5.5 combine

`array Set::combine ($data, $path1 = null, $path2 = null, $groupPath = null)`

Crée un tableau associatif avec $path1 comme chemin pour construire ses clés et éventuellement $path2 comme chemin pour récupérer les valeurs. Si $path2 n'est pas spécifié, toutes les valeurs seront initialisées à null (utile pour Set::merge). Facultativement, vous pouvez grouper les valeurs par ce qu'on obtient en suivant le chemin spécifié dans $groupPath.

```
$resultat = Set::combine(array(), '{n}.Utilisateur.id', '{n}.Utilisateur.Donnees');
// $resultat == array();

$resultat = Set::combine('', '{n}.Utilisateur.id', '{n}.Utilisateur.Donnees');
// $resultat == array();

$a = array(
    array(
        'Utilisateur' => array(
            'id' => 2,
            'groupe_id' => 1,
            'Donnees' => array(
                'utilisateur' => 'mariano.iglesias',
                'nom' => 'Mariano Iglesias'
            )
        )
    ),
    array(
        'Utilisateur' => array(
            'id' => 14,
            'groupe_id' => 2,
            'Donnees' => array(
                'utilisateur' => 'phpnut',
                'nom' => 'Larry E. Masters'
            )
        )
    ),
    array(
        'Utilisateur' => array(
            'id' => 25,
            'groupe_id' => 1,
            'Donnees' => array(
                'utilisateur' => 'gwoo',
                'nom' => 'The Gwoo'
            )
        )
    )
);
$resultat = Set::combine($a, '{n}.Utilisateur.id');
```

```
/* $resultat devrait ressembler à cela :
    Array
    (
        [2] =>
        [14] =>
        [25] =>
    )
*/

$resultat = Set::combine($a, '{n}.Utilisateur.id', '{n}.Utilisateur.inexistant');
/* $resultat devrait ressembler à cela :
    Array
    (
        [2] =>
        [14] =>
        [25] =>
    )
*/

$resultat = Set::combine($a, '{n}.Utilisateur.id', '{n}.Utilisateur.Donnees');
/* $resultat devrait ressembler à cela :
    Array
    (
        [2] => Array
            (
                [utilisateur] => mariano.iglesias
                [nom] => Mariano Iglesias
            )
        [14] => Array
            (
                [utilisateur] => phpnut
                [nom] => Larry E. Masters
            )
        [25] => Array
            (
                [utilisateur] => gwoo
                [nom] => The Gwoo
            )
    )
*/

$resultat = Set::combine($a, '{n}.Utilisateur.id', '{n}.Utilisateur.Donnees.nom');
/* $resultat devrait ressembler à cela :
    Array
    (
        [2] => Mariano Iglesias
        [14] => Larry E. Masters
        [25] => The Gwoo
    )
*/

$resultat = Set::combine($a, '{n}.Utilisateur.id', '{n}.Utilisateur.Donnees',
'{n}.Utilisateur.groupe_id');
/* $resultat devrait ressembler à cela :
    Array
    (
        [1] => Array
            (
                [2] => Array
                    (
                        [utilisateur] => mariano.iglesias
                        [nom] => Mariano Iglesias
                    )
                [25] => Array
                    (
```

```
                        [utilisateur] => gwoo
                        [nom] => The Gwoo
                    )
            )
        [2] => Array
            (
                [14] => Array
                    (
                        [utilisateur] => phpnut
                        [nom] => Larry E. Masters
                    )
            )
    )
*/

$resultat = Set::combine($a, '{n}.Utilisateur.id', '{n}.Utilisateur.Donnees.nom',
'{n}.Utilisateur.groupe_id');
/* $resultat devrait ressembler à cela :
    Array
    (
        [1] => Array
            (
                [2] => Mariano Iglesias
                [25] => The Gwoo
            )
        [2] => Array
            (
                [14] => Larry E. Masters
            )
    )
*/

$resultat = Set::combine($a, '{n}.Utilisateur.id', array('{0} : {1}',
'{n}.Utilisateur.Donnees.utilisateur', '{n}.Utilisateur.Donnees.nom'),
'{n}.Utilisateur.groupe_id');
/* $resultat devrait ressembler à cela :
    Array
    (
        [1] => Array
            (
                [2] => mariano.iglesias : Mariano Iglesias
                [25] => gwoo : The Gwoo
            )
        [2] => Array
            (
                [14] => phpnut : Larry E. Masters
            )
    )
*/

$resultat = Set::combine($a, array('{0} : {1}', '{n}.Utilisateur.Donnees.utilisateur',
'{n}.Utilisateur.Donnees.nom'), '{n}.Utilisateur.id');
/* $resultat devrait ressembler à cela :
    Array
    (
        [mariano.iglesias : Mariano Iglesias] => 2
        [phpnut : Larry E. Masters] => 14
        [gwoo : The Gwoo] => 25
    )
*/

$resultat = Set::combine($a, array('{1} : {0}', '{n}.Utilisateur.Donnees.utilisateur',
'{n}.Utilisateur.Donnees.nom'), '{n}.Utilisateur.id');
/* $resultat devrait ressembler à cela :
    Array
```

```
    (
        [Mariano Iglesias : mariano.iglesias] => 2
        [Larry E. Masters : phpnut] => 14
        [The Gwoo : gwoo] => 25
    )
*/

$resultat = Set::combine($a, array('%1$s : %2$d', '{n}.Utilisateur.Donnees.utilisateur',
'{n}.Utilisateur.id'), '{n}.Utilisateur.Donnees.nom');

/* $resultat devrait ressembler à cela :
    Array
    (
        [mariano.iglesias : 2] => Mariano Iglesias
        [phpnut : 14] => Larry E. Masters
        [gwoo : 25] => The Gwoo
    )
*/

$resultat = Set::combine($a, array('%2$d : %1$s', '{n}.Utilisateur.Donnees.utilisateur',
'{n}.Utilisateur.id'), '{n}.Utilisateur.Donnees.nom');
/* $resultat devrait ressembler à cela :
    Array
    (
        [2 : mariano.iglesias] => Mariano Iglesias
        [14 : phpnut] => Larry E. Masters
        [25 : gwoo] => The Gwoo
    )
*/
```

8.5.6 normalize

`array Set::normalize ($list, $assoc = true, $sep = ',', $trim = true)`

Normalise une chaîne ou une liste de tableau.

```
$a = array('Arbre', 'CompteurDeCache','Telechargement' => array(
            'repertoire' => 'produits',
            'champs' => array('image_1_id', 'image_2_id', 'image_3_id', 'image_4_id',
'image_5_id')
));
$b =  array('Cachable' => array('actif' => false),
        'Limite',
        'Liaison',
        'Validateur',
        'Transactionnel');
$resultat = Set::normalize($a);
/* $resultat ressemble maintenant à :
    Array
    (
        [Arbre] =>
        [CompteurDeCache] =>
        [Telechargement] => Array
            (
                [repertoire] => produits
                [champs] => Array
                    (
                        [0] => image_1_id
                        [1] => image_2_id
                        [2] => image_3_id
                        [3] => image_4_id
```

```
                        [4] => image_5_id
                    )
            )
    )
*/
$resultat = Set::normalize($b);
/* $resultat ressemble maintenant à :
    Array
    (
        [Cachable] => Array
            (
                [actif] =>
            )

        [Limite] =>
        [Liaison] =>
        [Validateur] =>
        [Transactionnel] =>
    )
*/
$resultat = Set::merge($a, $b); // Maintenant mixons les deux et normalisons
/* $resultat ressemble maintenant à :
    Array
    (
        [0] => Arbre
        [1] => CompteurDeCache
        [Telechargement] => Array
            (
                [repertoire] => produits
                [champs] => Array
                    (
                        [0] => image_1_id
                        [1] => image_2_id
                        [2] => image_3_id
                        [3] => image_4_id
                        [4] => image_5_id
                    )

            )
        [Cachable] => Array
            (
                [actif] =>
            )
        [2] => Limite
        [3] => Liaison
        [4] => Validateur
        [5] => Transactionnel
    )
*/
$resultat = Set::normalize(Set::merge($a, $b));
/* $resultat ressemble maintenant à :
    Array
    (
        [Arbre] =>
        [CompteurDeCache] =>
        [Telechargement] => Array
            (
                [repertoire] => products
                [champs] => Array
                    (
                        [0] => image_1_id
                        [1] => image_2_id
                        [2] => image_3_id
                        [3] => image_4_id
                        [4] => image_5_id
```

```
                    )

            )
        [Cachable] => Array
            (
                [actif] =>
            )
        [Limite] =>
        [Liaison] =>
        [Validateur] =>
        [Transactionnel] =>
    )
*/
```

8.5.7 countDim

`integer Set::countDim ($array = null, $all = false, $count = 0)`

Chiffre les dimensions d'un tableau. Si $all est défini comme false (qui est définit par défaut) il ne prendra en considération que la dimension du premier élément du 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' => '3.1.1.1')));
$result = Set::countDim($data);
// $result == 2

$data = array('1' => array('1.1' => '1.1.1'), '2', '3' => array('3.1' =>
array('3.1.1' => '3.1.1.1')));
$result = Set::countDim($data, true);
// $result == 3

$data = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' =>
array('2.1.1' => '2.1.1.1'))), '3' => array('3.1' => array('3.1.1' =>
'3.1.1.1')));
$result = Set::countDim($data, true);
// $result == 4

$data = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' =>
array('2.1.1' => array('2.1.1.1')))), '3' => array('3.1' => array('3.1.1' =>
'3.1.1.1')));
```

```
$result = Set::countDim($data, true);
// $result == 5

$data = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' =>
array('2.1.1' => array('2.1.1.1' => '2.1.1.1.1')))), '3' => array('3.1' =>
array('3.1.1' => '3.1.1.1')));
$result = Set::countDim($data, true);
// $result == 5

$set = array('1' => array('1.1' => '1.1.1'), array('2' => array('2.1' =>
array('2.1.1' => array('2.1.1.1' => '2.1.1.1.1')))), '3' => array('3.1' =>
array('3.1.1' => '3.1.1.1')));
$result = Set::countDim($set, false, 0);
// $result == 2

$result = Set::countDim($set, true);
// $result == 5
```

8.5.8 diff

array Set::diff ($val1, $val2 = null)

Calcule la différence entre un ensemble et un tableau, deux ensembles 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 donnera:
    Array
    (
        [2] => Array
            (
                [name] => contact
            )
    )
*/
$result = Set::diff($a, array());
/* $result donnera:
    Array
    (
        [0] => Array
            (
                [name] => main
            )
        [1] => Array
            (
                [name] => about
            )
    )
*/
$result = Set::diff(array(), $b);
/* $result donnera:
```

```
    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 donnera:
    Array
    (
        [0] => Array
            (
                [name] => main
            )
    )
*/
```

8.5.9 check

boolean/array Set::check ($data, $path = null)

Vérifie si un chemin particulier est défini dans un tableau. Si $path est vide, $data sera retournée plutôt qu'un booléen.

```
$set = array(
    'Mon Index 1' => array('Premier' => 'Le premier item')
);
$result = Set::check($set, 'Mon Index 1.Premier');
// $result == True
$result = Set::check($set, 'Mon Index 1');
// $result == True
$result = Set::check($set, array());
// $result == array('Mon Index 1' => array('Premier' => 'Le premier item'))
$set = array(
    'Mon Index 1' => array('Premier' =>
        array('Second' =>
            array('Troisieme' =>
                array('Quatrieme' => 'Lourd. Imbriqué.'))))
);
$result = Set::check($set, 'Mon Index 1.Premier.Second');
// $result == True
$result = Set::check($set, 'Mon Index 1.Premier.Second.Troisieme');
// $result == True
$result = Set::check($set, 'Mon Index 1.Premier.Second.Troisieme.Quatrieme');
// $result == True
```

```
$result = Set::check($set, 'Mon Index 1.Premier.Seconds.Troisieme.Quatrieme');
// $result == False
```

8.5.10 remove

array Set::remove ($list, $path = null)

Supprime un élément dans un ensemble ou un tableau tel que défini par $path.

```
$a = array(
    'pages'      => array('name' => 'page'),
    'files'         => array('name' => 'files')
);

$result = Set::remove($a, 'files');
/* $result donnera:
    Array
    (
        [pages] => Array
            (
                [name] => page
            )

    )
*/
```

8.5.11 classicExtract

array Set::classicExtract ($data, $path = null)

Obtient une valeur d'un tableau ou un objet qui est contenue dans un chemin donné en utilisant une syntaxe de chemin en tableau, à savoir:

- "{n}.Person.{[a-z]+}" - Ou "{n}" représente une clé numerique, "Person" représente une chaîne.
- "{[a-z]+}" (à savoir: toutes chaînes entre crochet comme {n} et {s} seront interprétées comme une expression 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 donnera:
    Array
    (
        [0] => 1
        [1] => 2
        [2] => 3
    )
*/
$result = Set::classicExtract($a, '{n}.Article.title');
/* $result donnera:
    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 donnera:
Array
    (
        [0] => Array
            (
                [0] => page
            )
        [1] => Array
            (
                [0] => fruit
            )
    )
*/
$result = Set::classicExtract($a, '{s}.{n}.name');
/* $result donnera:
    Array
    (
        [0] => Array
            (
                [0] => jippi
            )
        [1] => Array
            (
                [0] => jippi
            )
    )
*/
$result = Set::classicExtract($a,'{\w+}.{\w+}.name');
/* $result donnera:
    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 donnera:
    Array
    (
        [0] => Array
            (
                [pages] => page
            )
        [1] => Array
            (
                [fruites] => fruit
            )
    )
*/
$result = Set::classicExtract($a,'{n}.{\w+}.name');
/* $result donnera:
    Array
    (
        [0] => Array
            (
                [pages] => page
            )
        [1] => Array
            (
                [fruites] => fruit
            )
    )
*/
$result = Set::classicExtract($a,'{s}.{\d+}.name');
/* $result donnera:
    Array
    (
        [0] => Array
            (
                [0] => jippi
            )
        [1] => Array
            (
                [0] => jippi
            )
    )
*/
$result = Set::classicExtract($a,'{s}');
/* $result donnera:
    Array
    (

        [0] => Array
            (
                [0] => Array
                    (
                        [name] => jippi
                    )
            )
        [1] => Array
            (
                [0] => Array
                    (
```

```
                        [name] => jippi
                    )
            )
    )
*/
$result = Set::classicExtract($a,'{[a-z]}');
/* $result donnera:
    Array
    (
        [test] => Array
            (
                [0] => Array
                    (
                        [name] => jippi
                    )
            )

        [dot.test] => Array
            (
                [0] => Array
                    (
                        [name] => jippi
                    )
            )
    )
*/
$result = Set::classicExtract($a, '{dot\.test}.{n}');
/* $result donnera:
    Array
    (
        [dot.test] => Array
            (
                [0] => Array
                    (
                        [name] => jippi
                    )
            )
    )
*/
```

8.5.12 matches

boolean Set::matches ($conditions, $data=array(), $i = null, $length=null)

Set:: matchs peut être utilisé pour voir si un seul élément ou un XPath donné correspond à 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']);
// returns false
$res=Set::matches(array('id>=2'), $a[1]['Article']);
// returns true
$res=Set::matches(array('id>=3'), $a[1]['Article']);
// returns false
$res=Set::matches(array('id<=2'), $a[1]['Article']);
// returns true
$res=Set::matches(array('id<2'), $a[1]['Article']);
// returns false
```

```
$res=Set::matches(array('id>1'), $a[1]['Article']);
// returns true
$res=Set::matches(array('id>1', 'id<3', 'id!=0'), $a[1]['Article']);
// returns true
$res=Set::matches(array('3'), null, 3);
// returns true
$res=Set::matches(array('5'), null, 5);
// returns true
$res=Set::matches(array('id'), $a[1]['Article']);
// returns true
$res=Set::matches(array('id', 'title'), $a[1]['Article']);
// returns true
$res=Set::matches(array('non-existant'), $a[1]['Article']);
// returns false
$res=Set::matches('/Article[id=2]', $a);
// returns true
$res=Set::matches('/Article[id=4]', $a);
// returns false
$res=Set::matches(array(), $a);
// returns true
```

8.5.13 extract

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`array Set::extract ($path, $data=null, $options=array())`

Set::extract uses basic XPath 2.0 syntax to return subsets of your data from a find or a find all. This function allows you to retrieve your data quickly without having to loop through multi dimentional arrays or traverse through tree structures.

If $path is an array or $data is empty it the call is delegated to Set::classicExtract.

```
// Common Usage:
$users = $this->User->find("all");
$results = Set::extract('/User/id', $users);
// results returns:
// array(1,2,3,4,5,...);
```

Currently implemented selectors:

Selector	Note
/User/id	Similar to the classic {n}.User.id
/User[2]/name	Selects the name of the second User
/User[id<2]	Selects all Users with an id < 2
/User[id>2][<5]	Selects all Users with an id > 2 but < 5
/Post/Comment[author_name=john]/../name	Selects the name of all Posts that have at least one Comment written by john
/Posts[title]	Selects all Posts that have a 'title' key

/Comment/.[1]	Selects the contents of the first comment
/Comment/.[:last]	Selects the last comment
/Comment/.[:first]	Selects the first comment
/Comment[text=/cakephp/i]	Selects all comments that have a text matching the regex /cakephp/i
/Comment/@*	Selects the key names of all comments

Currently only absolute paths starting with a single '/' are supported. Please report any bugs as you find them. Suggestions for additional features are welcome.

To learn more about Set::extract() refer to function testExtract() in /cake/tests/cases/libs/set.test.php.

8.5.14 format

`array Set::format ($data, $format, $keys)`

Retourne une série de valeurs extraites d'un tableau, formatée en chaines.

```
$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}')));

$res = Set::format($data, '{1}, {0}', array('{n}.Person.first_name',
'{n}.Person.last_name'));
/*
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.something'));
```

```
/*
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.something'));
/*
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.something'));
/*
Array
(
    [0] => Nate, 42
    [1] => Larry, 0
    [2] => Garrett, 0
)
*/
```

8.5.15 enum

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`string Set::enum ($select, $list=null)`

The enum method works well when using html select elements. It returns a value from an array list if the key exists.

If a comma separated $list is passed arrays are numeric with the key of the first being 0 $list = 'no, yes' would translate to $list = array(0 => 'no', 1 => 'yes');

If an array is used, keys can be strings example: array('no' => 0, 'yes' => 1);

$list defaults to 0 = no 1 = yes if param is not passed

```
$res = Set::enum(1, 'one, two');
// $res is 'two'

$res = Set::enum('no', array('no' => 0, 'yes' => 1));
// $res is 0

$res = Set::enum('first', array('first' => 'one', 'second' => 'two'));
// $res is 'one'
```

8.5.16 numeric

`boolean Set::numeric ($array=null)`

Vérifie que toutes les valeurs du tableau sont numériques

```
    $data = array('un');
    $res = Set::numeric(array_keys($data));

    // $res est vrai

    $data = array(1 => 'un');
    $res = Set::numeric($data);

    // $res est faux

    $data = array('un');
    $res = Set::numeric($data);

    // $res est faux

    $data = array('un' => 'deux');
    $res = Set::numeric($data);

    // $res est faux

    $data = array('un' => 1);
    $res = Set::numeric($data);

    // $res est vrai

    $data = array(0);
    $res = Set::numeric($data);

    // $res est vrai

    $data = array('un', 'deux', 'trois', 'quatre', 'cinq');
    $res = Set::numeric(array_keys($data));

    // $res est vrai

    $data = array(1 => 'un', 2 => 'deux', 3 => 'trois', 4 => 'quatre', 5 =>
'cinq');
    $res = Set::numeric(array_keys($data));

    // $res est vrai

    $data = array('1' => 'un', 2 => 'deux', 3 => 'trois', 4 => 'quatre', 5 =>
'cinq');
    $res = Set::numeric(array_keys($data));

    // $res est vrai

    $data = array('un', 2 => 'deux', 3 => 'trois', 4 => 'quatre', 'a' => 'cinq');
    $res = Set::numeric(array_keys($data));

    // $res est faux
```

8.5.17 map

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`object Set::map ($class = 'stdClass', $tmp = 'stdClass')`

This method Maps the contents of the Set object to an object hierarchy while maintaining numeric keys as arrays of objects.

Basically, the map function turns array items into initialized class objects. By default it turns an array into a stdClass Object, however you can map values into any type of class. Example: 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 now looks like:

    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
            )

    )

*/
```

Using Set::map() with a custom class for second parameter:

```
class MyClass {
    function sayHi() {
        echo 'Hi!';
    }
}

$mapped = Set::map($data, 'MyClass');
//Now you can access all the properties as in the example above,
//but also you can call MyClass's methods
$mapped->[0]->sayHi();
```

8.5.18 pushDiff

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

`array Set::pushDiff ($array1, $array2)`

This function merges two arrays and pushes the differences in array2 to the bottom of the resultant array.

Example 1

```
$array1 = array('ModelOne' => array('id'=>1001, 'field_one'=>'a1.m1.f1',
'field_two'=>'a1.m1.f2'));
$array2 = array('ModelOne' => array('id'=>1003, 'field_one'=>'a3.m1.f1',
'field_two'=>'a3.m1.f2', 'field_three'=>'a3.m1.f3'));
$res = Set::pushDiff($array1, $array2);

/* $res now looks like:
    Array
    (
        [ModelOne] => Array
            (
                [id] => 1001
                [field_one] => a1.m1.f1
                [field_two] => a1.m1.f2
                [field_three] => a3.m1.f3
            )
    )
*/
```

Example 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 now looks like:
    Array
    (
        [a] => b
        [1] => 20938
        [c] => string
        [b] => b
        [3] => 238
        [4] => Array
            (
                [0] => extra_field
            )
    )
*/
```

8.5.19 filter

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

array Set::filter ($var, $isArray=null)

Filters empty elements out of a route array, excluding '0'.

```
$res = Set::filter(array('0', false, true, 0, array('one thing', 'I can tell you', 'is you
got to be', false)));

/* $res now looks like:
    Array (
        [0] => 0
        [2] => 1
        [3] => 0
        [4] => Array
            (
                [0] => one thing
                [1] => I can tell you
                [2] => is you got to be
                [3] =>
            )
    )
*/
```

8.5.20 merge

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

array Set::merge ($arr1, $arr2=null)

This function can be thought of as a hybrid between PHP's array_merge and array_merge_recursive. The difference to the two is that if an array key contains another array then the function behaves recursive (unlike array_merge) but does not do if for keys containing strings (unlike array_merge_recursive). See the unit test for more information.

This function will work with an unlimited amount of arguments and typecasts non-array parameters into arrays.

```
$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 now looks like:
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
)
*/
```

8.5.21 contains

`boolean Set::contains ($val1, $val2 = null)`

Détermine si un ensemble ou un tableau contient les clés et les valeurs exactes d'un autre tableau.

```
$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
```

8.6 Security

`Cache::read($key, $config = null)`

Cache::read() est utilisé pour lire la valeur mise en cache sous la clé `$key` issue de `$config`. Si $config est null la configuration par défaut sera utilisée. `Cache::read()` retournera la valeur en cache si elle est une valeur de cache valide ou `false` si le cache a expiré ou n'existe pas. Les contenus du cache pourraient être évalués à false, donc assurez-vous d'utiliser l'opérateur de comparaison stricte === ou `!==`.

Par exemple :

```
$nuage = Cache::read('nuage');

if ($nuage !== false) {
return $nuage;
}

// génère un nuage de données
// ...

// Stocke les données dans le cache
Cache::write('nuage', $nuage);
return $nuage;
```

8.7.2 Cache::write()

`Cache::write($key, $value, $config = null);`

Cache::write() va écrire une valeur $value dans le cache. Vous pouvez lire ou détruire cette valeur plus tard en vous référant à la clef `$key`. Vous pouvez spécifier également dans quelle configuration de cache vous voulez stocker cette valeur. Si aucune configuration `$config` est spécifiée, celle par défaut sera utilisée. Cache::write() peut stocker n'importe quel type d'objet et est idéal pour stocker les retours des méthodes find des modèles.

```
    if (($posts = Cache::read('posts')) === false) {
        $posts = $this->Post->find('all');
        Cache::write('posts', $posts);
    }
```

Cet exemple utilise Cache::write() et Cache::read() pour réduire le nombre d'aller-retour à la base de données pour rapatrier les posts.

8.7.3 Cache::delete()

`Cache::delete($key, $config = null)`

`Cache::delete()` vous permettra de supprimer complètement un objet stocké dans le magasin de Cache.

8.7.4 Cache::config()

`Cache::config()` est utilisé pour créer des configurations supplémentaires de Cache. Ces configurations supplémentaires peuvent avoir des durées, des moteurs, des chemins ou des préfixes différents de ceux de votre configuration de cache par défaut. Utiliser des configurations de cache multiples peut aider à réduire le nombre de fois où vous avez besoin d'utiliser `Cache::set()`, en plus de centraliser tous vos réglages de cache.

Vous devez spécifier quel moteur utiliser. Il n'utilise **pas** File par défaut.

```
Cache::config('court', array(
    'engine' => 'File',
    'duration'=> '+1 hours',
    'path' => CACHE,
    'prefix' => 'cake_court_'
));

// long
Cache::config('long', array(
    'engine' => 'File',
    'duration'=> '+1 week',
    'probability'=> 100,
    'path' => CACHE . 'long' . DS,
));
```

En plaçant le code ci-dessus dans votre `app/config/core.php` vous aurez deux configurations de Cache supplémentaires. Le nom de ces configurations, 'court' ou 'long', est utilisé comme paramètre `$config` pour `Cache::write()` et `Cache::read()`.

8.7.5 Cache::set()

`Cache::set()` vous permet de redéfinir de manière temporaire les paramètres de configuration du cache pour une opération (habituellement une lecture ou une écriture dans le cache). Si vous utilisez `Cache::set()` pour changer le paramétrage d'une opération d'écriture, vous devez également changer ce paramétrage avec `Cache::set()` avant de lire dans le cache la donnée écrite. Si vous omettez de le faire, c'est le paramétrage par défaut qui sera pris en compte.

```
Cache::set(array('duration' => '+30 days'));
Cache::write('results', $data);

// Plus loin

Cache::set(array('duration' => '+30 days'));
$results = Cache::read('results');
```

Si vous faites cet appel à `Cache::set()` très fréquemment, c'est qu'il est peut-être souhaitable de créer une nouvelle configuration du cache. Il ne sera alors plus nécessaire de faire appel à `Cache::set()`.

8.8 HttpSocket

La méthode get effectue une simple requête HTTP GET qui retourne les résultats.

`string get($uri, $query, $request)`

`$uri` est l'adresse web où la requête est effectuée, `$query` est n'importe quel paramètre de requête, soit sous la forme d'une chaîne : "param1=foo&param2=bar", soit sous la forme d'un tableau avec des clefs : array('param1' => 'foo', 'param2' => 'bar').

```
App::import('Core', 'HttpSocket');
$HttpSocket = new HttpSocket();
$resultats = $HttpSocket->get('http://www.google.com/search', 'q=cakephp');
//Retourne la version HTML des résultats de recherche Google pour la requête
"cakephp"
```

8.8.2 post

La méthode post effectue une simple requête HTTP POST et retourne les résultats.

`string function post ($uri, $data, $request)`

Les paramètres pour la méthode `post` sont pratiquement les mêmes que pour la méthode `get`, `$uri` est l'adresse web où la requête est faite, `$data` sont les données à POSTer, soit sous la forme d'une chaine : "param1=foo&param2=bar", soit sous la forme d'un tableau avec des clefs : array('param1' => 'foo', 'param2' => 'bar').

```
App::import('Core', 'HttpSocket');
$HttpSocket = new HttpSocket();
$resultats = $HttpSocket->post('www.unsite.com/ajouter', array('nom' => 'test',
'type' => 'utilisateur'));
//$resultats contient ce qui est retourné par le POST.
```

8.8.3 request

La méthode de base request est appelée depuis tous les conteneurs (get, post, put, delete). Il retourne le résultat de la requête.

`string function request($request)`

$request est un tableau associatif d'options diverses. Voici son format et ses paramètres par défaut :

```
var $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,
    'cookies' => array()
);
```

8.9 Router

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Router can be used to parse urls into arrays containing indexes for the controller, action, and any parameters, and the opposite: to convert url arrays (eg. array('controller'=>'posts', 'action'=>'index')) to string urls.

Read more about ways to configure the Router here: http://book.cakephp.org/view/945/Routes-Configuration

Router also include other utility methods for dealing with urls.

9 Applications en mode console intégrées

CakePHP offre un certain nombre d'applications en mode console prêtes à l'emploi. Certaines des ces applications sont utilisées conjointement avec d'autres fonctionnalités de CakePHP (comme ACL ou i18n) et d'autres sont faites pour un usage général en vous permettant de les lancer rapidement.

Cette rubrique explique comment utiliser les applications en mode console livrées avec CakePHP.

Avant que vous ne vous y plongiez, vous pourriez revoir la section Console CakePHP couverte précédemment. La configuration de la Console n'est pas expliquée ici, donc si vous ne l'avez jamais utilisée avant, relisez la section correspondante.

9.1 Génération de code avec Bake

Vous avez déjà découvert le prototypage (*scaffolding*) avec CakePHP : une façon simple de visualiser l'application finale avec seulement une base de données et quelques classes minimales. 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 : modèles, vues et contrôleurs. 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) pourraient trouver le *screencast* Bake 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).

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 créé auparavant.

Après que vous ayez créé un fichier de configuration de base de données, exécuter *Bake* vous présentera les options suivantes :

```
---------------------------------------------------------------
App : app
Path: /path-to/project/app
---------------------------------------------------------------
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
```

9.1.1 Bake improvements in 1.3

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

For 1.3 bake has had a significant overhaul, and a number of features and enhancements have been built in.

- Two new tasks (FixtureTask and TestTask) are accessible from the main bake menu
- A third task (TemplateTask) has been added for use in your shells.
- All the different bake tasks now allow you to use connections other than default for baking. Using the `-connection` parameter.
- Plugin support has been greatly improved. You can use either `-plugin PluginName` or `Plugin.class`.
- Questions have been clarified, and made easier to understand.
- Multiple validations on models has been added.
- Self Associated models using `parent_id` are now detected. For example if your model is named Thread, a ParentThread and ChildThread association will be created.
- Fixtures and Tests can be baked separately.
- Baked Tests include as many fixtures as they know about, including plugin detection (plugin detection does not work on PHP4).

So with the laundry list of features, we'll take some time to look at some of the new commands, new parameters and updated features.

New FixtureTask, TestTask and TemplateTask.

Fixture and test baking were a bit of a pain in the past. You could only generate tests when baking the classes, and fixtures could only be generated when baking models. This made adding tests to your applications later or even regenerating fixtures with new schemas a bit painful. For 1.3 we've separated out Fixture and Test making them separate tasks. This allows you to re-run them and regenerate tests and fixtures at any point in your development process.

In addition to being rebuildable at any time, baked tests are now attempt to find as many fixtures as possible. In the past getting into testing often involved fighting through numerous 'Missing Table' errors. With more advanced fixture detection we hope to make testing easier and more accessible.

Test cases also generate skeleton test methods for every non-inherited public method in your classes. Saving you one extra step.

`TemplateTask` is a behind the scenes task, and it handles file generation from templates. In previous versions of CakePHP baked views were template based, but all other code was not. With 1.3 almost all the content in the files generated by bake are controlled by templates and the `TemplateTask`.

The `FixtureTask` not only generates fixtures with dummy data, but using the interactive options or the `-records` option you can enable fixture generation using live data.

New bake command

New commands have been added to make baking easier and faster. Controller, Model, View baking all feature an `all` subcommand, that builds everything at once and makes speedy rebuilds easy.

```
cake bake model all
```

Would bake all the models for an application in one shot. Similarly `cake bake controller all` would bake all controllers and `cake bake view all` would generate all view files. Parameters on the `ControllerTask` have changed as well. `cake bake controller scaffold` is now `cake bake controller public`. `ViewTask` has had an `-admin` flag added, using `-admin` will allow you to bake views for actions that begin with `Routing.admin`

As mentioned before `cake bake fixture` and `cake bake test` are new, and have several subcommands each. `cake bake fixture all` will regenerate all the basic fixtures for your application. The `-count` parameter allows you to set the number of fake records that are created. By running fixture task interactively you can generate fixtures using the data in your live tables. You can use `cake bake test <type> <class>` to create test cases for already created objects in your app. Type should be one of the standard CakePHP types ('component', 'controller', 'model', 'helper', 'behavior') but doesn't have to be. Class should be an existing object of the chosen type.

Templates Galore

New in bake for 1.3 is the addition of more templates. In 1.2 baked views used templates that could be changed to modify the view files bake generated. In 1.3 templates are used to generate all output from bake. There are separate templates for controllers, controller action sets, fixtures, models, test cases, and the view files from 1.2. As well as more templates, you can also have multiple template sets or, bake themes. Bake themes can be provided in your app, or as part of plugins. An example plugin path for bake theme would be `app/plugins/bake_theme/vendors/shells/templates/dark_red/`. An app bake theme called `blue_bunny` would be placed in `app/vendors/shells/templates/blue_bunny`. You can look at `cake/console/templates/default/` to see what directories and files are required of a bake theme. However, like view files, if your bake theme doesn't implement a template, other installed themes will be checked until the correct template is found.

Additional plugin support.

New in 1.3 are additional ways to specify plugin names when using bake. In addition to `cake bake plugin Todo controller Posts`, there are two new forms. `cake bake controller Todo.Posts` and `cake bake controller Posts -plugin Todo`. The plugin parameter can be while using interactive bake as well. `cake bake controller -plugin Todo`, for example will allow you to use interactive bake to add controllers to your Todo plugin. Additional / multiple plugin paths are supported as well. In the past bake required your plugin to be in app/plugins. In 1.3 bake will find which of the pluginPaths the named plugin is located on, and add the files there.

9.2 Gestion du schéma et migrations

Un fichier de schéma généré vous permet de transporter facilement un schéma agnostique de votre base de données. Vous pouvez générer un fichier de schéma de votre base en utilisant la commande :

```
```

```
$ cake schema generate
```

Cela va créer un fichier schema.php dans votre dossier `app/config/schema` .

Le shell *schema* n'utilise que les tables pour lesquelles des modèles 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 :

```
$ cake schema run 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:

```
$ cake schema dump nomdufichier.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.

9.2.2 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 :

```
$ cake schema generate
```

Vous aurez alors les choix suivants :

```
Generating Schema...
Schema file exists.
 [O]verwrite
 [S]napshot
 [Q]uit
Would you like to do? (o/s/q)
```

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.

9.3 Modifier le rendu HTML produit par les templates de "bake"

Si vous voulez modifier le rendu HTML produit par défaut par la commande "bake", suivez ces simples étapes :

Pour "cuire" des vues personnalisées :

1. Allez dans : cake/console/templates/default/views
2. Remarquez les 4 fichiers qui s'y trouvent
3. Copiez les vers votre répertoire : app/vendors/shells/templates/[themename]/views
4. Changez le rendu HTML pour contrôler la façon dont "bake" construit vos vues.

Le segment de chemin `[themename]` devrait correspondre au nom du thème bake que vous êtes en train de créer. Les noms de thème bake doivent être uniques, donc n'utilisez pas 'default'.

Pour "cuire" des projets personnalisés :

1. Allez dans: cake/console/templates/skel
2. Remarquez les fichiers de base de l'application qui s'y trouvent
3. Copiez les dans votre répertoire : app/vendors/shells/templates/skel
4. Changez le rendu HTML pour contrôler la façon dont "bake" construit vos vues.

5. Passez le paramètre 'chemin du squelette' à la tâche project :
   ```
   cake bake project -skel vendors/shells/templates/skel
   ```

Notes

- Vous devez lancer la commande `cake bake project` pour que le paramètre path puisse être passé.
- Le chemin du template est relatif au chemin courant de l'Interface de Ligne de Commande (CLI).
- Vu que le chemin absolu vers le squelette nécessite d'être entré manuellement, vous pouvez spécifier n'importe quel répertoire contenant le template que vous voulez construire, y compris en utilisant plusieurs templates (sauf si Cake commence à gérer la surcharge du répertoire skel comme il le fait pour les vues)

10 Déploiement

Dans cette section, vous pouvez découvrir des applications CakePHP typiques pour voir comment toutes les pièces s'assemblent.

Sinon, vous pouvez vous référer à la Forge de CakePHP et à la Boulangerie pour des applications et composants existants. N'oubliez pas que vous pouvez également voir le code source de ce "livre de cuisine".

11.1 Le tutoriel du blog CakePHP

Bienvenue dans Cake ! Vous êtes probablement en train de lire ce tutoriel car vous voulez en savoir plus sur le fonctionnement de CakePHP. Notre but est d'augmenter votre productivité de rendre la programmation plus agréable : nous espérons que vous le remarquerez lorsque vous vous plongerez dans le code.

Ce tutoriel va vous guider à travers la création d'un blog simple. Nous allons obtenir et installer Cake, créer et configurer une base de données et créer suffisamment de logique applicative pour lister, ajouter, éditer et supprimer les articles du blog.

Voici ce dont vous aurez besoin :

1. **Un serveur web fonctionnel**. Nous supposerons que vous utilisez Apache, bien que les instructions pour l'utilisation 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**. Le plus vous aurez d'expérience en programmation orienté objet, le 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 chapître "Débuter avec CakePHP", section : Comprendre le modèle M-V-C . Ne vous inquiétez pas : il n'y a qu'une demi-page de lecture.

Maintenant, lançons-nous !

11.1.1 Obtenir Cake

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

Pour télécharger la dernière révision, visitez le projet CakePHP sur Cakeforge : http://cakeforge.org/projects/cakephp/ et téléchargez la version stable. Pour ce tutoriel, vous avez besoin de la version 1.2.x.x

Vous pouvez également faire un "checkout" ou un "export" SVN d'une copie récente de notre "trunk" à l'adresse suivante : https://svn.cakephp.org/repo/trunk/cake/1.2.x.x/

Quelque soit le moyen utilisé pour le télécharger, placez le code dans le "DocumentRoot" de votre serveur. Une fois terminé, votre répertoire d'installation devrait ressembler à celui-ci :

```
/chemin_du_document_root
    /app
    /cake
```

```
    /docs
    /vendors
    .htaccess
    index.php
```

A présent, il est peut-être temps de voir un peu comment fonctionne la structure de fichiers de Cake : lisez le chapitre "Principes de base de CakePHP", section Structure de fichiers dans CakePHP.

11.1.2 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 donnée vide du nom de votre choix. 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 sur 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 vraiment excitant, non ?', NOW());
```

Les choix des noms pour la table et les colonnes ne sont pas arbitraires. Si vous respectez les conventions de nommage Cake pour la base de données et les classes (toutes deux expliquées ici : Conventions CakePHP), vous tirerez avantage d'un grand nombre de fonctionnalités automatiques et vous éviterez des étapes de configuration. Cake est suffisamment souple pour implémenter les pires schémas de base de données, mais respecter les conventions vous fera gagner du temps.

Consultez le chapître Conventions CakePHP pour plus d'information, mais il suffit de dire que nommer notre table 'posts' permet de la relier automatiquement à notre modèle Post, et que les champs nommés 'modified' et 'created' seront automatiquement gérés par Cake.

11.1.3 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 la dernière fois que vous configurerez quelque chose.

Une copie du fichier de configuration de CakePHP 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 :

```
var $default = array(
    'driver' => 'mysql',
    'persistent' => 'false',
    'host' => 'localhost',
    'port' => '',
    'login' => 'cakeBlog',
    'password' => 'c4k3-rUl3Z',
    'database' => 'cake_blog_tutorial',
    'schema' => '',
    'prefix' => '',
    'encoding' => ''
);
```

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

11.1.4 Configuration facultative

Il y a deux autres éléments qui peuvent être configurés. La plupart des développeurs configurent les éléments de cette petite liste, mais cela n'est pas requis pour ce tutoriel. Le premier point consiste à définir une chaîne de caractères personnalisée (ou "grain de sel") afin de sécuriser les hashs. Le second point est de permettre l'accès en écriture à CakePHP pour son dossier `tmp`.

Le "grain" de sécurité est utilisé pour générer des hashs. Changez sa valeur par défaut en éditant : `/app/config/core.php` à la ligne 151. La nouvelle valeur n'a pas beaucoup d'importance, du moment qu'elle n'est pas facile à deviner.

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

La dernière étape consiste à rendre le répertoire `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 les propriétés du répertoire `app/tmp` pour cet utilisateur. La commande finale 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.

11.1.5 Une note sur mod_rewrite

De temps en temps, un nouvel utilisateur rencontrera des problèmes avec *mod_rewrite*, je vais donc les mentionner ici en marge. Si la page d'accueil de CakePHP vous semble un peu singulière (pas d'images ou de style CSS), cela signifie probablement que *mod_rewrite* n'est pas activé sur votre système. Voici quelques conseils pour vous aider à le faire fonctionner :

1. Assurez-vous qu'une neutralisation (*override*) .htaccess est permise : dans votre fichier `httpd.conf`, vous devriez avoir une rubrique qui définit une section pour chaque répertoire de votre serveur. Vérifiez que `AllowOverride` est défini à `All` pour le bon répertoire.
2. Assurez-vous que vous éditez le bon `httpd.conf` et non celui d'un utilisateur ou d'un site spécifique.
3. Pour une raison ou une autre, vous avez peut être téléchargé une copie de CakePHP sans les fichiers .htaccess nécessaires. Cela arrive parfois car certains systèmes d'exploitation masquent les fichiers qui commencent par '.' et ne les copient pas. Assurez vous que votre copie de CakePHP provient de la section téléchargements du site ou de notre dépôt SVN.
4. Assurez-vous qu'Apache charge correctement le *mod_rewrite* ! Vous devriez voir quelque chose comme : `LoadModule rewrite_module libexec/httpd/mod_rewrite.so` et `AddModule mod_rewrite.c` dans votre `httpd.conf`.

Si vous ne voulez pas ou ne pouvez pas faire fonctionner le *mod_rewrite* (ou tout autre module compatible) sur votre serveur, vous devrez utiliser les "URLs enjolivées" intégrées à Cake. Dans `/app/config/core.php`, décommentez la ligne qui ressemble à cela :

```
Configure::write('App.baseUrl', env('SCRIPT_NAME'));
```

Supprimez également ces fichiers .htaccess :

```
        /.htaccess
        /app/.htaccess
        /app/webroot/.htaccess
```

Vos URLs seront ainsi transformées en : www.example.com/index.php/controllername/actionname/param plutôt que www.example.com/controllername/actionname/param.

11.1.6 Créer un Modèle "Post"

La classe Modèle c'est le pain quotidien des applications CakePHP. En créant un modèle 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 Modèle de CakePHP se placent dans `/app/models`, et le fichier que nous allons créer maintenant sera enregistré dans `/app/models/post.php`. Le fichier complet devrait ressembler à ceci :

```
<?php

class Post extends AppModel
{
    var $name = 'Post';
}

?>
```

La convention de nommage est très importante dans CakePHP. En nommant notre modèle "Post", CakePHP peut automatiquement déduire que ce modèle sera utilisé dans le Contrôleur "Posts" et qu'il sera lié à une table de la base de données appelée `posts`.

CakePHP créé un modèle automatiquement s'il ne trouve pas de fichier correspondant dans **`/app/models`**. En clair, si vous faites une erreur de nommage accidentelle (i.e. **`Post.php`** ou **`posts.php`**) CakePHP n'utilisera pas vos paramètres et les remplacera par ceux par défaut.

C'est toujours une bonne idée d'ajouter la variable $name, elle est en effet utilisée pour surmonter quelques bizarreries dans les noms des classes en PHP4.

Pour plus d'informations sur les modèles, comme les préfixes des tables, les callbacks et la validation, consultez le chapitre Modèles du manuel.

11.1.7 Créer un Contrôleur "Posts"

Nous allons maintenant créer un contrôleur pour nos posts. Le contrôleur est l'endroit où s'exécutera toute la logique métier pour l'interaction du processus de post. En un mot, c'est l'endroit où vous jouez avec les modèles et où les tâches liées aux posts s'exécutent. Nous placerons ce nouveau contrôleur dans un fichier appelé `posts_controller.php` au sein du répertoire `/app/controllers`. Voici à quoi devrait ressembler le contrôleur de base :

```
<?php
class PostsController extends AppController {
    var $name = 'Posts';
}
?>
```

A présent, ajoutons une action à notre contrôleur. Les actions représentent souvent une simple fonction ou une interface dans une application. Par exemple, lorsque les utilisateurs requêtent la page www.exemple.com/posts/index (ce qui est équivalent à 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 çà :

```
<?php
class PostsController extends AppController {

    var $name = 'Posts';

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

Laissez-moi vous expliquer un peu cette action. En définissant la fonction index() dans notre Contrôleur "Posts", 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 nommée foobar(), les utilisateurs auraient la possibilité d'accéder à www.exemple.com/posts/foobar.

Note: vous pourriez être tenté de nommer vos contrôleurs et vos actions d'une certaine manière pour obtenir une certaine URL. Résistez à cette tentation. Suivez les conventions CakePHP (le nom des contrôleurs 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 contrôleur 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 modèle Post. Notre modèle Post est automatiquement disponible via `$this->Post`, parce que nous avons suivi les conventions de nommage de Cake.

Pour en apprendre plus sur les contrôleurs de Cake, consultez notre chapitre "Développer avec CakePHP" à la section : "Contrôleurs".

11.1.8 Créer les Vues Post

Maintenant que nous avons nos données en provenance du modèle, ainsi que la logique applicative et les flux définis par notre contrôleur, nous allons créer une vue pour l'action "index" que nous avons créée 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) retourne :

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] => Un titre encore une fois
                    [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 vraiment 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 contrôleur (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 :

```
/app/views/posts/index.ctp

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

    <!-- C'est ici que nous bouclons sur le tableau $posts afin d'afficher les
informations des posts -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo $html->link($post['Post']['title'],
"/posts/view/".$post['Post']['id']); ?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
    <?php endforeach; ?>

</table>
```

Bien entendu, cela donnera quelque chose de simple.

Vous devez avoir remarqué l'utilisation d'un objet appelé `$html`. C'est une instance de la classe `HtmlHelper` de CakePHP. 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 de formulaires, du JavaScript et de l'Ajax. Vous pouvez en apprendre plus sur la manière de les utiliser dans le chapitre "Helpers intégrés", 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, vous donnez simplement un chemin relatif à partir de la base de l'application et Cake s'occupe du reste. En tant que tel, vos URLs prendront généralement la forme suivante : `/controleur/action/parametre1/parametre2`.

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 sÃ»rement 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 Contrôleur Posts :

```
<?php
class PostsController extends AppController {

    var $name = 'Posts';

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

    function view($id = null) {
        $this->Post->id = $id;
        $this->set('post', $this->Post->read());
    }
}
?>
```

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 post unique.

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

Maintenant, créons la vue pour notre nouvelle action "view" et plaçons la dans : `/app/views/posts/view.ctp`.

```
/app/views/posts/view.ctp

<h1><?php echo $post['Post']['title']?></h1>

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

<p><?php echo $post['Post']['body']?></p>
```

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

11.1.9 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.

D'abord, commençons par créer une action `add()` dans le Contrôleur Posts :

```
<?php
class PostsController extends AppController {
    var $name = 'Posts';

    function index() {
```

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

    function view($id) {
        $this->Post->id = $id;
        $this->set('post', $this->Post->read());

    }

    function add() {
        if (!empty($this->data)) {
            if ($this->Post->save($this->data)) {
                $this->flash('Votre post a été sauvegardé.','/posts');
            }
        }
    }
}
?>
```

Voici ce que fait l'action `add()` : si le formulaire de données envoyé n'est pas vide, nous essayons d'enregistrer les données en utilisant le modèle Post. Si pour une raison quelconque cela ne s'enregistre pas, nous effectuons juste un rendu de la vue. Cela nous donne une chance de voir les erreurs de validation de l'utilisateur et d'autres alertes.

Lorsqu'un utilisateur utilise un formulaire pour POSTER des données à votre application, cette information est disponible dans `$this->data`. Vous pouvez vous servir de `pr()` pour l'afficher si vous souhaitez voir à quoi cela ressemble.

La fonction `$this->flash()` appelée ici est une méthode du contrôleur qui affiche un message à l'utilisateur pendant une seconde (en utilisant la mise en page des messages flashs), puis redirige l'utilisateur vers une autre URL (`/posts`, dans ce cas). Si DEBUG est paramétré à 0, `$this->flash()` redirigera automatiquement, au contraire, si DEBUG est > à 0, vous serez en mesure de voir la mise en page des flashs et de cliquer sur le message pour réaliser la redirection.

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.

11.1.10 Validation des données

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

Pour tirer avantage des fonctionnalités de validation, vous devrez utiliser le Helper "Form" de Cake dans vos vues. Le Helper "Form" est disponible, par défaut, pour toutes les vues, avec la variable `$form`.

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

```
/app/views/posts/add.ctp

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

Ici, nous utilisons le Helper "Form" pour générer la balise d'ouverture d'un formulaire HTML. Voici le code HTML produit par `$form->create()` :

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

Si `create()` est appelé sans aucun paramètre, on suppose que vous construisez un formulaire qui envoie les données à l'action `add()` du contrôleur courant, via POST.

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

L'appel de `$form->end()` génère un bouton de soumission et termine le formulaire. Si une chaîne de caractères est passée comme premier paramètre de la méthode `end()`, le Helper "Form" affiche un bouton de soumission dont le nom correspond à celle-ci, ainsi que la balise de fermeture du formulaire. Encore une fois, référez-vous au Chapitre "Helpers intégrés" pour en savoir plus sur les helpers.

Si vous le souhaitez, vous pouvez mettre à jour votre vue `/app/views/posts/index.ctp` pour y inclure un nouveau lien "Ajouter un post" qui pointe vers www.exemple.com/posts/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 modèle. Retournons donc à notre modèle Post et faisons quelques ajustements :

```
<?php
class Post extends AppModel
{
    var $name = 'Post';

    var $validate = array(
        'title' => array(
            'rule' => array('minLength', 1)
        ),
        'body' => array(
            'rule' => array('minLength', 1)
        )
    );
}
?>
```

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, dispose d'un certain nombre de règles pré-fabriquées (codes de carte bancaire, adresses emails, etc.) et d'une souplesse pour la personnalisation des règles de validation. Pour plus d'informations sur cette configuration, consultez le chapitre sur la validation des données.

Maintenant que vous avez mis en place vos règles de validation, lancez l'application pour essayer d'ajouter un post avec un titre ou un contenu vide, afin de voir comment cela fonctionne. Puisque nous avons utilisé la méthode input() du Helper "Form" pour créer nos éléments de formulaire, nos messages d'erreurs de validation seront affichés automatiquement.

11.1.11 Supprimer des Posts

A présent, mettons en place un moyen de suppression des billets pour les utilisateurs. Démarrons avec une action `delete()` dans le PostsController :

```
function delete($id) {
    $this->Post->delete($id);
    $this->flash('Le post avec l\'id: '.$id.' a été supprimé.',
'/posts');
}
```

Cette logique supprime le billet spécifié par "$id" et utilise `flash()` pour afficher à l'utilisateur un message de confirmation avant de le rediriger vers /posts.

Parce 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 qui permettent aux utilisateurs de supprimer des billets, ainsi :

```
/app/views/posts/index.ctp

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

<!-- C'est ici que nous bouclons sur le tableau $posts afin d'afficher les informations
des posts -->

    <?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
        <?php echo $html->link($post['Post']['title'],
'/posts/view/'.$post['Post']['id']);?>
        </td>
        <td>
        <?php echo $html->link('Supprimer', "/posts/delete/{$post['Post']['id']}", null,
'Etes-vous sÃ»r ?' )?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
    <?php endforeach; ?>

</table>
```

Note : le code de cette vue utilise également le Helper "Html" pour afficher à l'utilisateur un message de confirmation JavaScript avant qu'il ne tente de supprimer un billet.

11.1.12 Editer des Posts

Edition de posts : allons-y ! Vous êtes un pro de CakePHP maintenant, vous devriez donc avoir adopté le principe. Créer l'action puis la vue. Voici à quoi devrait ressembler l'action `edit()` du Contrôleur Posts :

```
function edit($id = null) {
    if (empty($this->data)) {
        $this->Post->id = $id;
        $this->data = $this->Post->read();
    } else {
        if ($this->Post->save($this->data['Post'])) {
            $this->flash('Votre post a été mis à jour.','/posts');
        }
    }
}
```

Cette action contrôle d'abord les données soumises par le formulaire. Si rien n'a été envoyé, elle trouve le post et transmet les données à la vue. Si des données *ont* été transmises, elle essaye d'enregistrer les données en utilisant le modèle Post (ou retourne en arrière et affiche à l'utilisateur les erreurs de validation).

La vue "edit" devrait ressembler à quelque chose comme cela :

```
/app/views/posts/edit.ctp

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

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

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

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

```
/app/views/posts/index.ctp (lien d'édition ajouté)

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

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

<?php foreach ($posts as $post): ?>
    <tr>
        <td><?php echo $post['Post']['id']; ?></td>
        <td>
            <?php echo
$html->link($post['Post']['title'],'/posts/view/'.$post['Post']['id']);?>
```

```
                </td>
                <td>
            <?php echo $html->link('Supprimer', "/posts/delete/{$post['Post']['id']}",
null, 'Etes-vous sÃ»r ?')?>
            <?php echo $html->link('Editer', '/posts/edit/'.$post['Post']['id']);?>
        </td>
        <td><?php echo $post['Post']['created']; ?></td>
    </tr>
<?php endforeach; ?>

</table>
```

11.1.13 Routes

Maintenant, abordons les 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 recherche apprécieront de comprendre comment lier des URLs à des appels spécifiques de fonction dans CakePHP. Nous allons juste faire une rapide modification des routes dans ce tutoriel. Pour plus d'informations sur les techniques avancées de routage, consultez le chapitre "Développer avec CakePHP" section: "Configuration des routes".

Pour le moment, CakePHP redirigera une personne visitant la racine de votre site (c'est-à-dire http://www.exemple.com) vers le Contrôleur Pages et rentourne une vue appelée "home". Au lieu de cela, nous voudrions que les utilisateurs de notre blog soient redirigés vers notre Contrôleur Posts.

Le routage de Cake se trouve dans `/app/config/routes.php`. Vous devrez commenter ou supprimer la ligne qui définit la route par défaut de la racine. 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 contrôleur, ajoutons donc une ligne ressemblant à ceci :

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

Cela devrait connecter les utilisateurs demandant '/' à l'action index() de notre Contrôleur Posts fraîchement créé.

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 une 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 vers la même destination.

11.1.14 Conclusion

Créer des applications de cette manière vous apportera la paix, l'honneur, des femmes (ou des hommes) et de l'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 souhaités 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. Lancez votre propre projet, lisez le reste du Manuel et l'API.

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

Prochaines lectures suggérrées

Voici les prochains sujets sur lesquels se penchent le plus souvent les apprentis cuisiniers :

1. Gabarits : Personnaliser les Gabarits (Layouts) de votre application
2. Eléments : Inclure et ré-utiliser les portions de vues
3. Scaffolding : Construire une ébauche d'application sans avoir à coder
4. Utiliser Bake : Générer un code CRUD basique
5. Authentification : Enregistrement et connexion d'utilisateurs

11.2 Application simple contrôlée par Acl

Bienvenue dans *CakePHP* si vous n'avez jamais utilisé *CakePHP* commencer par le tutoriel sur le blog. Si vous avez déjà regardé le tutoriel du blog et que vous avez lu des choses à propos de *bake* (l'utilitaire en ligne de commande de *CakePHP*) ou que vous l'avez utilisé, et si vous voulez mettre en place un système d'authentification et de controle d'accès alors ce tutoriel est pour vous.

Comme mentionné ci-dessus, ce tutoriel suppose que vous avez une certaine expérience de *CakePHP*. Vous êtes familier de tous les concepts MVC du coeur de Cake. Vous êtes aussi à l'aise pour utiliser *bake* et la console de *cake*. Si ce n'est pas le cas, vous devriez apprendre ces concepts et reprendre ce tutoriel après. Ce tutoriel sera ainsi plus facile à suivre et à comprendre. Dans ce tutoriel, nous utiliserons le Component Auth et le Component Acl. Si vous ne savez pas ce qu'ils sont, regardez leurs pages dans le CookBook avant de continuer.

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 .

   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.
2. 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.

11.2.1 Préparation de notre application

Premièrement, récupérons une copie fraîche de CakePHP

Pour obtenir un téléchargement à jour, visitez le projet CakePHP chez CakeForge: http://cakeforge.org/projects/cakephp/ et téléchargez la version stable. Pour ce tutoriel vous aurez besoin de la 1.2.x.x

Vous pouvez aussi faire une extraction/un export SVN du *trunk* de notre code à : https://svn.cakephp.org/repo/trunk/cake/1.2.x.x/

Une fois que vous avez votre copie toute fraîche 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 utilisateurs (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    pseudo VARCHAR(255) NOT NULL UNIQUE,
    mot_passe CHAR(40) NOT NULL,
    groupe_id INT(11) NOT NULL,
    created DATETIME,
    modified DATETIME
);


CREATE TABLE groupes (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(100) NOT NULL,
    created DATETIME,
    modified DATETIME
);


CREATE TABLE posts (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    utilisateur_id INT(11) NOT NULL,
    titre VARCHAR(255) NOT NULL,
    contenu TEXT,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE gadgets (
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(100) NOT NULL,
    numero VARCHAR(12),
    quantite 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 *cake bake* pour créer rapidement vos modèles, contrôleurs et vues. Ne vous occupez pas des routes "admin" pour l'instant, c'est un sujet déjà bien compliqué sans elles. Assurez-vous également de **ne pas** ajouter le composant Acl, ni le composant Auth à l'un ou l'autre de vos contrôleurs pendant que vous les créez. Nous nous occuperons de cela bientôt. Vous devriez maintenant avoir des modèles, des contrôleurs et des vues pour les utilisateurs, les groupes, les posts et les gadgets.

Dans le mode `actions`, cela désigne nos ARO (Objets de requêtes d'accès) - les groupes et les utilisateurs - opposés aux objets ACO (Objets de contrôle d'accès) - les contrôleurs & actions.

11.2.2 Préparation pour ajouter 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 composants Auth et Acl. Tout d'abord, ajoutez une action login et une action logout à votre `UtilisateursController`.

```
function login() {
    // Magie du composant Auth
}

function logout() {
    // Laissez vide pour le moment.
}
```

Ensuite, créez le fichier de vue suivant pour le login, dans app/views/utilisateurs/login.ctp :

```
$session->flash('auth');
echo $form->create('Utilisateur', array('action' => 'login'));
echo $form->inputs(array(
    'legend' => __('Identification', true),
    'username',
    'password'
));
echo $form->end('Identifier');
```

Une fois que c'est configuré correctement, nous n'avons pas besoin de nous inquiéter d'ajouter quoi que ce soit pour hacher les mots de passe, puisque le composant Auth le fera pour nous automatiquement à la création/édition des utilisateurs et lors de leur identification. De plus, si vous hachez les mots de passe entrants manuellement, le composant Auth ne sera pas capable de vous identifier. Car il les hachera à nouveau et qu'ils ne correspondront pas.

Ensuite, nous devons faire quelques modifications dans `AppController`. Si vous n'avez pas `/app/app_controller.php`, créez-le. Notez que cela doit aller dans **`/app/`** et non pas dans **`/app/controllers/`**. Du fait que nous souhaitons contrôler tout notre site avec Auth et Acl, nous allons les définir dans `AppController`.

```
class AppController extends Controller {
    var $components = array('Acl', 'Auth');

    function beforeFilter() {
        // Configuration de AuthComponent
        $this->Auth->userModel = 'Utilisateur';
        $this->Auth->authorize = 'actions';
        $this->Auth->loginAction = array('controller' => 'utilisateurs', 'action'
=> 'login');
        $this->Auth->logoutRedirect = array('controller' => 'utilisateurs',
'action' => 'login');
        $this->Auth->loginRedirect = array('controller' => 'posts', 'action'
=> 'add');
    }
}
```

Avant de mettre en place l'ACL, nous aurons besoin d'ajouter quelques utilisateurs et groupes. Avec l'utilisation de `AuthComponent`, nous ne serons pas en mesure d'accéder à l'une de nos actions, puisque nous ne sommes pas connectés. Nous allons maintenant ajouter quelques exceptions, ainsi `AuthComponent` nous permettra de créer quelques groupes et utilisateurs. Dans **chacun** de vos `GroupesController` et `UtilisateursController`, ajoutez ce qui suit :

```
function beforeFilter() {
    parent::beforeFilter();
    $this->Auth->allowedActions = array('*');
```

```
}
```

Ces déclarations indiquent au composant Auth qu'il doit permettre un accès public à toutes les actions. C'est seulement temporaire et ce sera supprimé une fois que nous aurons quelques utilisateurs et groupes dans notre base de données. N'ajoutez pas d'utilisateurs ou de groupes pour le moment.

11.2.3 Initialiser les tables Acl dans la BdD

Avant de créer des utilisateurs et groupes, 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. Pour supprimer ces erreurs, nous devons exécuter un fichier de schéma. Dans un shell, exécutez la commande suivante : `cake schema create DbAcl`. Ce schéma vous invite à supprimer et créer les tables. Répondez Oui (Yes) à la suppression et création des tables.

Pensez à spécifier le chemin du dossier de l'application si vous êtes en dehors de celui-ci.

1. Dans votre dossier d'application:

   ```
   $ /chemin/vers/cake/console/cake schema create DbAcl
   ```
2. En dehors de votre dossier d'application :

   ```
   $ /chemin/vers/cake/console/cake -app /chemin/vers/dossier/app schema create
   DbAcl
   ```

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/appli/cake/console/templates/skel/config/schema/db_acl.sql.

Avec les contrôleurs 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 modèles utilisateurs et groupes. Concrètement, faire qu'ils s'attachent auto-magiquement à l'Acl.

11.2.4 Agir comme un Requêteur

Pour que Auth et Acl fonctionnent correctement, nous devons associer nos utilisateurs et groupes dans les entrées de nos tables Acl. Pour ce faire, nous allons utiliser le comportement `AclBehavior`. Le comportement `AclBehavior` permet de connecter automagiquement des modèles avec l'Acl. Son utilisation requiert l'implémentation de `parentNode()` dans vos modèles. Dans notre Modèle `Utilisateur` nous allons ajouter le code suivant :

```
var $name = 'Utilisateur';
var $belongsTo = array('Groupe');
var $actsAs = array('Acl' => array('requester'));

function parentNode() {
    if (!$this->id && empty($this->data)) {
        return null;
    }
    $data = $this->data;
    if (empty($this->data)) {
        $data = $this->read();
    }
    if (!$data['Utilisateur']['groupe_id']) {
        return null;
    } else {
```

```
        return array('Groupe' => array('id' => $data['Utilisateur']['groupe_id']));
    }
}
```

Ensuite dans notre Modèle `Groupe` ajoutons ce qui suit :

```
var $actsAs = array('Acl' => array('requester'));

function parentNode() {
    return null;
}
```

Cela permet de lier les modèles `Groupe` et `Utilisateur` à l'Acl, et de dire à CakePHP que chaque fois que l'on créé un Utilisateur ou un Groupe, 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 `utilisateurs` et `groupes`. Ainsi, chaque fois que vous créez ou supprimez un groupe/utilisateur, la table Aro est mise à jour.

Nos contrôleurs et modèles sont maintenant prêts à recevoir des données initiales et nos modèles `Groupe` et `Utilisateur` sont reliés à la table Acl. Ajoutez donc quelques groupes et utilisateurs en utilisant les formulaires créés avec Bake. J'ai créé les groupes suivants :

- administrateurs
- managers
- utilisateurs

J'ai également créé un utilisateur dans chaque groupe, de façon à avoir un utilisateur 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 | Groupe       |           1 | NULL  |    1 |    4 |
|  2 |      NULL | Groupe       |           2 | NULL  |    5 |    8 |
|  3 |      NULL | Groupe       |           3 | NULL  |    9 |   12 |
|  4 |         1 | Utilisateur  |           1 | NULL  |    2 |    3 |
|  5 |         2 | Utilisateur  |           2 | NULL  |    6 |    7 |
|  6 |         3 | Utilisateur  |           3 | NULL  |   10 |   11 |
+----+-----------+--------------+-------------+-------+------+------+
6 rows in set (0.00 sec)
```

Cela nous montre que nous avons 3 groupes et 3 utilisateurs. Les utilisateurs sont imbriqués dans les groupes, ce qui signifie que nous pouvons définir des permissions sur une base par groupe ou par utilisateur.

Lorsque l'on modifie l'appartenance d'un utilisateur à un groupe, vous devez mettre à jour l'ARO manuellement. Ce code doit être exécuté lorsque l'on met à jour les informations de l'utilisateur :

```
// Vérifie si le groupe de permission change
$anciengroupeid = $this->Utilisateur->field('groupe_id');
if ($anciengroupeid !== $this->data['Utilisateur']['groupe_id']) {
    $aro =& $this->Acl->Aro;
```

```
    $utilisateur = $aro->findByForeignKeyAndModel($this->data['Utilisateur']['id'],
'Utilisateur');
    $groupe =
$aro->findByForeignKeyAndModel($this->data['Utilisateur']['groupe_id'], 'Groupe');

    // Sauvegarde de la table ARO
    $aro->id = $utilisateur['Aro']['id'];
    $aro->save(array('parent_id' => $groupe['Aro']['id']));
}
```

11.2.4.1 ACL basé sur les groupe uniquement

Dans la cas où nous souhaiterions simplifier en utilisant les permissions par groupes, nous avons besoin d'implémenter `bindNode()` dans le modèle `User`.

```
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 utilisateur devra être assigné avec 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)
```

11.2.5 Créer les ACOs

Maintenant que nous avons nos utilisateur et groupes (aros), nous pouvons commencer à intégrer nos contrôleurs existants dans l'Acl et définir des permissions pour nos groupes et utilisateurs, et permettre la connexion / déconnexion.

Nos AROs sont automatiquement créés lorsque de nouveaux utilisateurs et groupes sont ajoutés. Qu'en est-t'il de l'auto-génération des ACOs pour nos contrôleurs 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'`AclComposant`. Créer les Acos depuis le shell ressemble à cela :

```
cake acl create aco root controllers
```

En utilisant l'AclComposant, cela ressemblera à :

```
$this->Acl->Aco->create(array('parent_id' => null, 'alias' =>
```

```
'controleurs'));
$this->Acl->Aco->save();
```

Ces deux exemples vont créer notre *root* ou ACO de plus haut niveau, qui sera appelé 'controleurs'. L'objectif de ce noeud *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 contrôleurs/actions, tels que la vérification des permissions d'un enregistrement d'un modèle. Puisque nous allons utiliser un ACO *root* global, nous devons faire une petite modification à la configuration de l'`AuthComposant`. L'`AuthComposant` doit être renseigné sur l'existence de ce noeud *root*, de sorte que lors des contrôles de l'ACL, le composant puisse utiliser le bon chemin de noeud lors de la recherche contrôleurs/actions. Dans l'`AppController`, ajouter ce qui suit à `beforeFilter` :

```
$this->Auth->actionPath = 'controleurs/';
```

11.2.6 Un outil automatique pour créer les ACOs

Comme mentionné précédemment, il n'y a pas de méthode toute faite pour importer tous vos contrôleurs et toutes vos actions dans l'Acl. Cependant, nous détestons tous faire des choses répétitives, comme saisir au clavier des centaines d'actions dans une grosse application. J'ai cuisiné une fonction automatique pour construire ma table Aco. Cette fonction regardera dans chaque contrôleur de votre application. Elle ajoutera toutes les méthodes non-privées et non spécifiques du `Contrôleur` dans la table Acl, rangées gentiment sous le nom de leur contrôleur. Vous pouvez ajouter et exécuter ceci dans votre `AppController` ou tout autre contrôleur d'ailleurs, assurez-vous simplement de le supprimer avant de mettre en production votre application.

```
/**
 * Reconstruit l'Acl à partir des contrôleurs actuels de l'application
 *
 * @return void
 */
    function buildAcl() {
        $log = array();

        $aco =& $this->Acl->Aco;
        $root = $aco->node('controllers');
        if (!$root) {
            $aco->create(array('parent_id' => null, 'model' => null, 'alias'
=> 'controllers'));
            $root = $aco->save();
            $root['Aco']['id'] = $aco->id;
            $log[] = 'Noeud Aco créé pour les contrôleurs';
        } else {
            $root = $root[0];
        }

        App::import('Core', 'File');
        $controleurs = Configure::listObjects('controller');
        $appIndex = array_search('App', $controleurs);
        if ($appIndex !== false ) {
            unset($controleurs[$appIndex]);
        }
        $methodes_de_base = get_class_methods('Controller');
        $methodes_de_base[] = 'buildAcl';

        // regarde dans chaque contrôleur de app/controllers
        foreach ($controleurs as $nomCtrl) {
```

```
            App::import('Controller', $nomCtrl);
            $classectrl = $nomCtrl . 'Controller';
            $methodes = get_class_methods($classectrl);

            // trouve / crée le noeud contrôleur
            $noeudControleur = $aco->node('controllers/' . $nomCtrl);
            if (!$noeudControleur) {
                $aco->create(array('parent_id' => $root['Aco']['id'], 'model' =>
null, 'alias' => $nomCtrl));
                $noeudControleur = $aco->save();
                $noeudControleur['Aco']['id'] = $aco->id;
                $log[] = 'Noeud Aco créé pour '. $nomCtrl;
            } else {
                $noeudControleur = $noeudControleur[0];
            }

            // nettoie les méthodes. pour supprimer celles qui sont dans Controller
et les actions privées.
            foreach ($methodes as $k => $methode) {
                if (strpos($methode, '_', 0) === 0) {
                    unset($methodes[$k]);
                    continue;
                }
                if (in_array($methode, $methodes_de_base)) {
                    unset($methodes[$k]);
                    continue;
                }
                $noeudMethode = $aco->node('controllers/' . $nomCtrl . '/' . $methode);
                if (!$noeudMethode) {
                    $aco->create(array('parent_id' => $noeudControleur['Aco']['id'],
'model' => null, 'alias' => $methode));
                    $noeudMethode = $aco->save();
                    $log[] = 'Noeud Aco créé pour '. $methode;
                }
            }
        }
        debug($log);
    }
```

Maintenant lancez l'action dans votre navigateur, comme çà : **`http://localhost/groups/buildacl`**, Ceci construira votre table ACO.

Vous pourriez vouloir conserver cette fonction sous la main, puisqu'elle ajoutera les nouveaux ACOs pour tous les contrôleurs et actions qui sont dans votre application, chaque fois que vous l'exécuterez. Elle ne supprime pas les noeuds pour les actions qui n'existent plus. Maintenant que le plus gros du travail est effectué, nous devons définir quelques permissions et supprimer le code qui désactivait le composant `Auth` au départ.

Ensuite, une fois que vous avez réalisé ce travail, il se peut que vous constatiez des problèmes d'accès aux plugins que vous utilisiez. Le truc pour automatiser les ACOs de contrôleur des plugins, c'est que le App::import doit suivre la convention pour le nommage de plugin NomPlugin.NomControleurPlugin.

Donc ce dont nous avons besoin, c'est d'une fonction qui nous donnera une liste des noms de contrôleur du plugin et de l'importer de la même manière que nous l'avons fait pour les contrôleur normaux, dans le code ci-dessus. La fonction ci-dessous réalisera justement cela :

```
/**
 * Obtenir les noms des contrôleurs du plugin
 *
```

```
 * Cette fonction prendra un tableau des noms de contrôleurs du plugin et
 * et s'assurera également que les contrôleurs sont disponibles pour nous
permettre d'obtenir
 * les noms des méthodes, en faisant un App::import pour chaque contrôleur du
plugin.
 *
 * @return array Liste des noms de plugin
 */
    function _getPluginControllerNames(){
        App::import('Core', 'File', 'Folder');
        $chemins = Configure::getInstance();
        $dossier =& new Folder();
        // change le répertoire pour celui des plugins
        $dossier->cd(APP . 'plugins');
        // récupère la liste des fichiers qui ont un nom se terminant par
controller.php
        $fichiers = $dossier->findRecursive('.*_controller\.php');
        // Récupère la liste des plugins
        $plugins = Configure::listObjects('plugin');

        // boucle sur les contrôleurs que nous avons trouvés dans le
répertoire des plugins
        foreach ($fichiers as $f => $nomFichier) {
            // récupère le nom de base du fichier
            $fichier = basename($nomFichier);
            // récupère le nom du contrôleur
            $fichier = Inflector::camelize(substr($fichier, 0, strlen($fichier) -
strlen('_controller.php')));

            // boucle sur les plugins
            foreach ($plugins as $nomPlugin) {
                if (preg_match('/^' . $nomPlugin . '/', $fichier)){
                    // premièrement, on se débarrasse du contrôleur
App du plugin
                    // nous faisons cela parce que le contrôleur App n'est jamais
appelé directement
                    if (preg_match('/^' . $nomPlugin . 'App/', $fichier)) {
                        unset($fichiers[$f]);
                    } else {
                        if (!App::import('Controller', $nomPlugin . '.' . $fichier)) {
                            debug('Erreur durant l'import de ' . $fichier . ' pour le
plugin ' . $nomPlugin);
                        }
                        // maintenant on prépare le nom du Plugin
                        // ceci est nécessaire pour nous permettre de
récupérer les noms de méthode
                        $fichiers[$f] = $fichier;
                    }
                    break;
                }
            }
        }
        return $fichiers;
    }
```

Vous pouvez alors, soit modifier le code original pour inclure les contrôleurs du plugin, en les mergant avec la liste que vous aviez (placez ceci avant la boucle foreach):

```
$plugins = $this->_getPluginControllerNames();
$controleurs = array_merge($controleurs, $plugins);
```

11.2.7 Définir 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 :

```
cake acl grant $aroAlias $acoAlias [create|read|update|delete|'*']
```

* doit être entouré de guillemets ('*')

Pour autoriser avec `AclComponent` faites comme suit :

```
$this->Acl->allow($aroAlias, $acoAlias);
```

Nous allons maintenant ajouter quelques déclarations d'autorisation/interdiction. Ajoutez ce qui suit dans une fonction temporaire de votre contrôleur `UtilisateursController` et rendez-vous depuis votre navigateur à l'adresse pour l'exécuter. Si vous faites un `SELECT * FROM aros_acos` vous devriez voir une pile entière de 0 et de 1. Une fois que vous avez vérifié que vos permissions sont définies, supprimez la fonction.

```
function initDB() {
    $groupe =& $this->Utilisateur->Groupe;
    // Autorise les admins à tout faire
    $groupe->id = 1;
    $this->Acl->allow($groupe, 'controllers');

    // On autorise les responsables des billets (posts) et des widgets (gadgets)
    $groupe->id = 2;
    $this->Acl->deny($groupe, 'controllers');
    $this->Acl->allow($groupe, 'controllers/Posts');
    $this->Acl->allow($groupe, 'controllers/Gadgets');

    // On autorise aux utilisateurs seulement l'ajout et la modification de billets et
gadgets
    $groupe->id = 3;
    $this->Acl->deny($groupe, 'controllers');
    $this->Acl->allow($groupe, 'controllers/Posts/add');
    $this->Acl->allow($groupe, 'controllers/Posts/edit');
    $this->Acl->allow($groupe, 'controllers/Gadgets/add');
    $this->Acl->allow($groupe, 'controllers/Gadgets/edit');
}
```

Nous avons désormais configuré quelques règles basiques d'accès. Nous avons autorisé les administrateurs à tout faire. Les responsables ont accès à tout ce qui concerne les billets et gadgets. Alors que les utilisateurs ne peuvent accéder qu'à l'ajout et la modification de billets et de gadgets.

Nous avons du créer une référence au modèle `Groupe` et modifier son id pour pouvoir spécifier l'ARO que nous voulions. Ceci est dÃ» au fonctionnement du comportement `Acl`. `AclBehavior` ne fixe pas le champ alias dans la table `aros`, donc nous devons utiliser une référence d'objet ou un tableau pour référencer l'ARO que nous voulons.

Vous avez pu remarqué que nous avions délibérément écarté index et voir des permissions Acl. Nous allons rendre les actions voir et index publiques dans `PostsController` et `GadgetsController`. Cela permet aux utilisateurs non-autorisés de voir ces pages, ce qui en fait des pages publiques. Cependant, vous pouvez à tout moment supprimer ces actions de `AuthComponent::allowedActions` et les permissions pour voir et index redeviendront celles des Acl.

Maintenant nous voulons extraire la référence à `Auth->allowedActions` dans nos contrôleurs utilisateurs et groupes. Ensuite ajoutez ce qui suit à vos contrôleurs Posts et Gadgets :

```
function beforeFilter() {
    parent::beforeFilter();
    $this->Auth->allowedActions = array('index', 'voir');
}
```

Cela enlève le "basculement à off" que nous avions mis plus tôt dans les contrôleurs utilisateurs et groupes et cela rend public l'accès aux actions index et voir dans les contrôleurs Posts et Gadgets. Dans `AppController::beforeFilter()` ajoutez ce qui suit :

```
 $this->Auth->allowedActions = array('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.

11.2.8 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/views/utilisateurs/login.ctp` si vous ne l'avez pas déjà fait.

```
<h2>Connexion</h2>
<?php
echo $form->create('Utilisateur', array('url' => array('controller' =>
'utilisateurs', 'action' =>'login')));
echo $form->input('Utilisateur.pseudo');
echo $form->input('Utilisateur.motdepasse');
echo $form->end('Connexion');
?>
```

Si l'utilisateur est déjà connecté, on le redirige :

```
   function login() {
    if ($this->Session->read('Auth.Utilisateur')) {
        $this->Session->setFlash('Vous êtes déjà connecté!');
        $this->redirect('/', null, false);
    }
}
```

Vous pouvez également vouloir ajouter dans votre mise en page un flash() pour les messages d'Auth. Copiez la mise en page par défaut du coeur - trouvable dans `cake/libs/views/layouts/default.ctp` - dans le

dossier layouts de votre application si vous ne l'avez pas encore fait. Dans `app/views/layouts/default.ctp` ajoutez

```
echo $this->Session->flash('auth');
```

Vous devriez maintenant pouvoir vous connecter et tout devrait fonctionner auto-magiquement. Quand l'accès est refusé, les messages d'Auth seront affichés si vous avez ajouté le code `$session->flash('auth')`

11.2.9 Déconnexion

Abordons maintenant la déconnexion. Nous avions plus tôt laissé cette fonction vide, il est maintenant temps de la remplir. Dans `UtilisateursController::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'Utilisateur 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.

11.2.10 C'est fini

Ce guide résume les nombreux changements nécessaires à la migration de CakePHP version 1.2 vers la version 1.3. Chaque section contient des informations pertinentes sur les modifications apportées aux méthodes existantes, ainsi que sur toutes les méthodes qui ont été supprimées / renommées.

Remplacement de fichiers (important)

- webroot/index.php : doit être remplacé suite aux changements effectués dans le processus de bootstrap.
- config/core.php : des paramètres supplémentaires ont été mis en place car nécessaires pour PHP 5.3.
- webroot/test.php : remplacer si vous voulez exécuter des tests unitaires.

Constantes supprimées

Les constantes suivantes ont été supprimées de CakePHP. Si votre application dépend de celles-ci, vous devez les redéfinir dans `app/config/bootstrap.php`

Configuration and application bootstrapping

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Bootstrapping Additional Paths.

In your app/config/bootstrap.php you may have variables like `$pluginPaths` or `$controllerPaths`. There is a new way to add those paths. As of 1.3 RC1 the `$pluginPaths` variables will no longer work. You must use
`App::build()` to modify paths.

```
App::build(array(
    'plugins' => array('/full/path/to/plugins/', '/next/full/path/to/plugins/'),
    'models' =>  array('/full/path/to/models/', '/next/full/path/to/models/'),
    'views' => array('/full/path/to/views/', '/next/full/path/to/views/'),
    'controllers' => array('/full/path/to/controllers/',
'/next/full/path/to/controllers/'),
    'datasources' => array('/full/path/to/datasources/',
'/next/full/path/to/datasources/'),
    'behaviors' => array('/full/path/to/behaviors/', '/next/full/path/to/behaviors/'),
    'components' => array('/full/path/to/components/',
'/next/full/path/to/components/'),
    'helpers' => array('/full/path/to/helpers/', '/next/full/path/to/helpers/'),
    'vendors' => array('/full/path/to/vendors/', '/next/full/path/to/vendors/'),
    'shells' => array('/full/path/to/shells/', '/next/full/path/to/shells/'),
    'locales' => array('/full/path/to/locale/', '/next/full/path/to/locale/'),
    'libs' => array('/full/path/to/libs/', '/next/full/path/to/libs/')
));
```

Also changed is the order in which bootstrapping occurs. In the past `app/config/core.php` was loaded **after** `app/config/bootstrap.php`. This caused any `App::import()` in an application bootstrap to be un-cached and considerably slower than a cached include. In 1.3 core.php is loaded and the core cache configs are created **before** bootstrap.php is loaded.

Loading custom inflections

`inflections.php` has been removed, it was an unnecessary file hit, and the related features have been refactored into a method to increase their flexibility. You now use `Inflector::rules()` to load custom inflections.

```
Inflector::rules('singular', array(
    'rules' => array('/^(bil)er$/i' => '\1', '/^(inflec|contribu)tors$/i' =>
'\1ta'),
    'uninflected' => array('singulars'),
    'irregular' => array('spins' => 'spinor')
));
```

Will merge the supplied rules into the infection sets, with the added rules taking precedence over the core rules.

Fichiers renommés et changements internes

Librairies renommées

Les librairies suivantes du coeur de CakePHP : libs/session.php, libs/socket.php, libs/model/schema.php et libs/model/behavior.php ont été renommées, afin d'obtenir une meilleure cohérence entre les noms de fichiers et les classes qui y sont incluses (ainsi que pour régler certains problèmes avec les espaces de nom) :

- 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 des fichiers ci-dessus n'affectera en rien la manière de coder.

L'héritage de la classe Object

Les classes suivantes n'étendent plus la classe Object :

- Router
- Set
- Inflector
- Cache
- CacheEngine

Si vous utilisiez des méthodes de la classe Object depuis ces classes, vous ne devrez plus utiliser ces méthodes.

Contrôleurs et Composants

Contrôleurs

`$title_for_layout` quand le layout est rendu. `$title_for_layout` est toujours rempli par défaut. Mais si vous voulez personnaliser cela, utilisez `$this->set('title_for_layout', $var)`. `$this->set('title_for_layout', $var);` à la place.
- Le Contrôleur a deux nouvelles méthodes : `startupProcess` et `shutdownProcess`. Ces méthodes sont responsables de la manipulation des processus de démarrage et d'arrêt du contrôleur.

Composant

Composant Cookie

Composant RequestHandler

`getReferer`

Composant Session

Assistant Session et Composant Session

Le second paramètre utilisé pour `SessionComponent::setFlash()`, sert à définir le layout et créer un fichier de rendu. Ceci a été modifié pour utiliser un `element`. Si vous spécifiez des gabarits de messages "session flash" personnalisés dans vos applications, vous devrez apporter les modifications suivantes :

1. Déplacez les fichiers de layouts requis dans app/views/elements
2. Renommez la variable $content_for_layout en $message

Le `Helper Session` et le `Composant Session` ne sont plus inclus automatiquement si vous ne faites pas appel à eux. Ils se comportent maintenant comme n'importe quel autre composant et doivent être déclarés comme n'importe quel autre composant/assistant. Vous devriez mettre à jour `AppController::$components` et `AppController::$helpers` afin d'y inclure ces classes pour maintenir le comportement de votre application.

```
var $components = array('Session', 'Auth', ...);
var $helpers = array('Session', 'Html', 'Form' ...);
```

Ces changements ont été faits pour rendre CakePHP plus explicite et déclaratif au niveau des classes que vous, le développeur d'application, voulez utiliser. Auparavant, il n'existait aucune façon d'éviter le chargement automatique des classes Session sans modifier les fichiers du coeur de CakePHP. Ce qui est une chose que nous vous permettre d'éviter. Par ailleurs, les classes de Session étaient les seuls composants et assistants magiques. Cette modification aide à uniformiser et normaliser le comportement entre toutes les classes.

Library Classes

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

CakeSession

SessionComponent

Folder

Set

String

statically.

Router

`Routing.admin` is deprecated. It provided an inconsistent behavior with other prefix style routes in that it was treated differently. Instead you should use `Routing.prefixes`. Prefix routes in 1.3 do not require additional routes to be declared manually. All prefix routes will be generated automatically. To update simply change your core.php.

```
//from:
Configure::write('Routing.admin', 'admin');

//to:
Configure::write('Routing.prefixes', array('admin'));
```

See the New features guide for more information on using prefix routes. A small change has also been done to routing params. Routed params should now only consist of alphanumeric chars, - and _ or `/[A-Z0-9-_+]+/`.

```
Router::connect('/:$%@#param/:action/*', array(...)); // BAD
Router::connect('/:can/:anybody/:see/:m-3/*', array(...)); //Acceptable
```

For 1.3 the internals of the Router were heavily refactored to increase performance and reduce code clutter. The side effect of this is two seldom used features were removed, as they were problematic and buggy even with the existing code base. First path segments using full regular expressions was removed. You can no longer create routes like

```
Router::connect('/([0-9]+)-p-(.*)/', array('controller' => 'products', 'action' =>
'show'));
```

These routes complicated route compilation and impossible to reverse route. If you need routes like this, it is recommended that you use route parameters with capture patterns. Next mid-route greedy star support has been removed. It was previously possible to use a greedy star in the middle of a 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

Also, people using the 'id' key in array-form URLs will notice that Router::url() now treats this as a named parameter. If you previously used this approach for passing the ID parameter to actions, you will need to rewrite all your $html->link() and $this->redirect() calls to reflect this change.

```
// old format:
$url = array('controller' => 'posts', 'action' => 'view', 'id' => $id);
// use cases:
Router::url($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` is no longer capable of setting a controller's layout/viewPath with request parameters. Control of these properties should be handled by the Controller, not the Dispatcher. This feature was also undocumented, and untested.

Debugger

`Debugger::checkSecurityKeys()`
- Calling `Debugger::output("text")` no longer works. Use `Debugger::output("txt")`.

Object

`CakeLog::write` is now called statically. See New Logging features for more information on changes made to logging.

Sanitize

escaped strings. In the past using the `$remove` parameter would skip entity encoding, returning possibly dangerous content. `remove_html` option. This will trigger the `strip_tags` feature of `Sanitize::html()`,

and must be used in conjunction with the `encode` parameter.

Configure and App

- Configure::listObjects() replaced by App::objects()
- Configure::corePaths() replaced by App::core()
- Configure::buildPaths() replaced by App::build()
- Configure no longer manages paths.
- Configure::write('modelPaths', array...) replaced by App::build(array('models' => array...))
- Configure::read('modelPaths') replaced by App::path('models')
- There is no longer a debug = 3. The 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.

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 New features guide 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

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Model

been removed in favor of `Model::delete()`, which is now the canonical delete method. 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()`.

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.

- 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

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

View

`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`

`$title_for_layout` when rendering the layout. `$title_for_layout` is still populated by default. But if you want to customize it, use `$this->set('title_for_layout', $var)`. `$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

`truncate()` instead.
- an `$highlighter` parameter. Use `$options['format']` instead.
- an `$considerHtml`parameter. Use `$options['html']` instead.
- 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

`$showEmpty` parameter. Use `$attributes['empty']` instead. `$showEmpty` parameter. Use `$attributes['empty']` instead. `$showEmpty` parameter. Use `$attributes['empty']` instead. `$showEmpty` parameter. Use `$attributes['empty']` instead. `$showEmpty` parameter. Use `$attributes['empty']` instead. `$showEmpty` parameter. Use `$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.

inputs other than type=submit. Use the type option to control the type of input generated. `<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::create()` no longer create hidden fieldset elements. Instead they create hidden div elements. This improves validation with HTML4.

Also be sure to check the Form helper improvements for additional changes and new features in the FormHelper.

HtmlHelper

`$inline` parameter. It has been merged with the `$options` array. `$escapeTitle` parameter. Use `$options['escape']` instead. `$escape` parameter. Use `$options['escape']` instead. `$escape` parameter. Use `$options['escape']` instead. `$escape` parameter. Use `$options['escape']` instead. `$inline` parameter. Use `$options['inline']` instead.

SessionHelper

`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 its 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

`$html->script()`

- `$javascript->codeBlock()` with `$html->scriptBlock()` or `$html->scriptStart()` and `$html->scriptEnd()` depending on your usage.

Console and shells

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Shell

`Shell::getAdmin()` has been moved up to `ProjectTask::getAdmin()`

Schema shell

```
cake schema create 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

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

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.

12.2 Nouveautés avec CakePHP 1.3

SecurityComponent

Les méthodes de la forme requireXX comme `requireGet` et `requirePost` acceptent maintenant un tableau comme argument aussi bien qu'une liste de chaînes de caractères.

```
$this->Security->requirePost(array('edit', 'update'));
```

Paramètres du composant

Les paramètres de tous les composants du *core* peuvent maintenant être édités depuis le tableau `$components`. Comme pour les comportements, vous pouvez déclarer les paramètres des composants lorsque vous déclarez le composant.

```
var $components = array(
    'Cookie' => array(
        'name' => 'MonCookie'
    ),
    'Auth' => array(
        'userModel' => 'MonUtilisateur',
        'loginAction' => array('controller' => 'utilisateurs', 'action' =>
'login')
    )
);
```

Cela devrait réduire l'encombrement dans les méthodes `beforeFilter()` de vos contrôleurs.

EmailComponent

- Vous pouvez maintenant retrouver le rendu de vos Emails envoyés, en lisant `$this->Email->htmlMessage` et `$this->Email->textMessage`. Ces propriétés contiendront l'email après rendu correspondant à leur nom.
- Plusieurs méthodes privées du composant Email ont été passée à *protected* pour une extension plus facile.
- La propriété `EmailComponent::$messageId` a été ajoutée. Elle permet un contrôle sur l'entête *Message-ID* pour les emails.

View & Helpers

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Helpers can now be addressed at `$this->Helper->func()` in addition to `$helper->func()`. This allows view variables and helpers to share names and not create collisions.

New JsHelper and new features in HtmlHelper

See JsHelper documentation for more information

Pagination Helper

Pagination helper provides additional css classes for styling and you can set the default sort() direction. `PaginatorHelper::next()` and `PaginatorHelper::prev()` now generate span tags by default, instead of divs.

Helper

`Helper::assetTimestamp()` has been added. It will add timestamps to any asset under WWW_ROOT. It works with `Configure::read('Asset.timestamp');` just as before, but the functionality used in Html and Javascript helpers has been made available to all helpers. Assuming `Asset.timestamp == force`

```
$path = 'css/cake.generic.css'
$stamped = $this->Html->assetTimestamp($path);
```

```
//$stamped contains 'css/cake.generic.css?5632934892'
```

The appended timestamp contains the last modification time of the file. Since this method is defined in `Helper` it is available to all subclasses.

TextHelper

highlight() now accepts an array of words to highlight.

NumberHelper

A new method `addFormat()` has been added. This method allows you to set currency parameter sets, so you don't have to retype them.

```
$this->Number->addFormat('NOK', array('before' => 'Kr. '));
$formatted = $this->Number->currency(1000, 'NOK');
```

FormHelper

The form helper has had a number of improvements and API modifications, see Form Helper improvements for more information.

Journalisation (logging)

La journalisation (logging) et `CakeLog` ont été considérablement améliorés, tous deux en fonctionnalités et en flexibilité. Voyez Nouvelles fonctionnalités de Logging pour plus d'informations.

Caching

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Cache engines have been made more flexible in 1.3. You can now provide custom `Cache` adapters in `app/libs` as well as in plugins using `$plugin/libs`. App/plugin cache engines can also override the core engines. Cache adapters must be in a cache directory. If you had a cache engine named `MyCustomCacheEngine` it would be placed in either `app/libs/cache/my_custom_cache.php` as an app/libs. Or in `$plugin/libs/cache/my_custom_cache.php` as part of a plugin. Cache configs from plugins need to use the plugin dot syntax.

```
Cache::config('custom', array(
    'engine' => 'CachePack.MyCustomCache',
    ...
));
```

App and Plugin cache engines should be configured in `app/bootstrap.php`. If you try to configure them in core.php they will not work correctly.

New Cache methods

Cache has a few new methods for 1.3 which make introspection and testing teardown easier.

Cache engine keys. engine. Once dropped cache engines are no longer readable or writeable. a numeric value. This is not implemented in FileEngine. a numeric value. This is not implemented in FileEngine.

Models, Behaviors and Datasource

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

App::import(), datasources & datasources from plugins

Datasources can now be included loaded with `App::import()` and be included in plugins! To include a datasource in your plugin you put it in `my_plugin/models/datasources/your_datasource.php`. To import a Datasource from a plugin use `App::import('Datasource', 'MyPlugin.YourDatasource');`

Using plugin datasources in your database.php

You can use plugin datasources by setting the datasource key with the plugin name. For example if you had a WebservicePack plugin with a LastFm datasource (plugin/webservice_pack/models/datasources/last_fm.php), you could do:

```
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

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.

tables. tables. tables.

In addition to tableParameters MySQL dbo's implement `fieldParameters`. fieldParameters allow you to control MySQL specific settings per column.

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

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Bake

Bake has had a number of significant changes made to it. Those changes are detailed in the bake updates section

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

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

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' => true));
$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' => false));
$this->Html->link('Go', array('controller' => 'posts', 'action' => 'index',
'admin' => false));
```

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

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

Inflector

You can now globally customize the default transliteration map used in Inflector::slug using Inflector::rules. eg. `Inflector::rules('transliteration', array('/Ã¥/' => 'aa', '/Ã¸/' => '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 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

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

Miscellaneous

Il n'y a pas encore de traduction pour cette section. Aidez-nous s'il vous plaît et traduire ceci.. Plus d'information à propos des traductions

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 {
    function phone($check) {
        ...
    }
    function postal($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.

```
var $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

Validation of IP Addresses has been extended to allow strict validation of a specific IP Version. It will also make use of PHP native validation mechanisms if available.

```
Validation::ip($someAddress);          // Validates both IPv4 and IPv6
Validation::ip($someAddress, 'IPv4'); // Validates IPv4 Addresses only
Validation::ip($someAddress, 'IPv6'); // Validates IPv6 Addresses only
```

Validation::uuid()

A uuid() pattern validation has been added to the `Validation` class. It will check that a given string matches a uuid by pattern only. It does not ensure uniqueness of the given uuid.