Symfony

The Book

for Symfony master

generated on August 31, 2013

Contents at a Glance

Chapter 1
Les fondamentaux de Symfony2 et HTTP

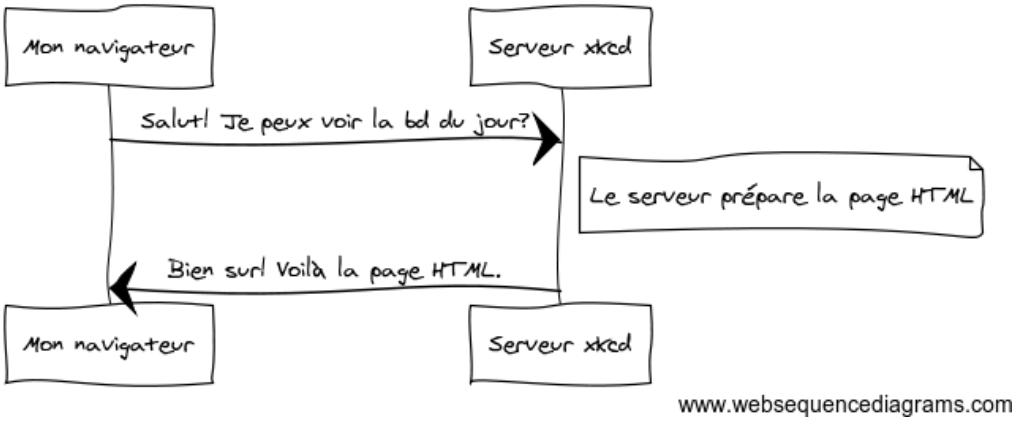
Félicitations! Grâce à l'apprentissage de Symfony2, vous êtes sur la bonne voie pour devenir un développeur web plus *productif* et *populaire* (en fait la popularité ne dépend que de vous). Symfony2 est construit de manière à revenir à l'essentiel : implémenter des outils qui vous aident à développer plus rapidement et à construire des applications plus robustes, sans pour autant vous gêner dans votre progression. Symfony repose sur les meilleures idées provenant de diverses technologies : les outils et concepts que vous êtes sur le point d'apprendre représentent les efforts de milliers de personnes depuis de nombreuses années. En d'autres termes, vous n'apprenez pas juste "Symfony", vous apprenez les fondamentaux du web, les bonnes pratiques de développement, et comment utiliser de nombreuses nouvelles bibliothèques PHP, internes ou indépendantes de Symfony2. Alors, soyez prêt !

Fidèle à la philosophie de Symfony2, ce chapitre débute par une explication du concept fondamental du développement web : HTTP. Quelles que soient vos connaissances ou votre langage de programmation préféré, ce chapitre **doit être lu** par tout un chacun.

HTTP est simple

HTTP (HyperText Transfer Protocol pour les geeks) est un langage texte qui permet à deux machines de communiquer ensemble. C'est tout ! Par exemple, lorsque vous regardez la dernière BD de *xkcd*[1], la conversation suivante (approximative) se déroule:

1. `http://xkcd.com/`



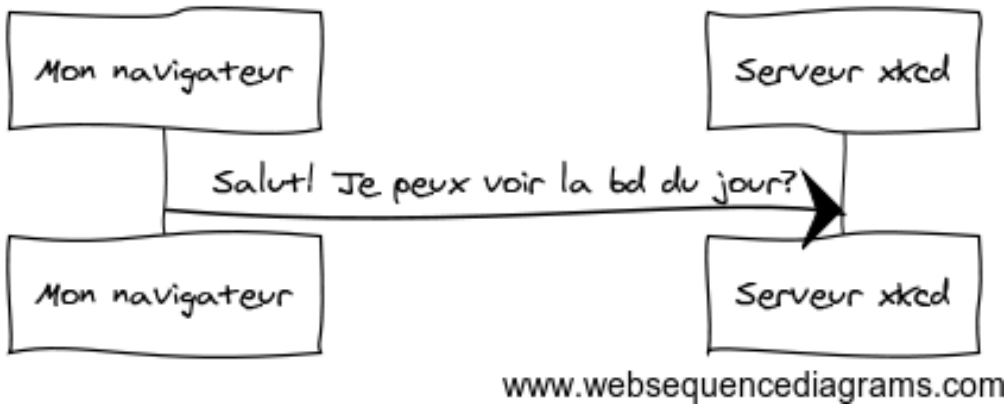
Et alors que l'actuel langage utilisé est un peu plus formel, cela reste toujours très simple. HTTP est le terme utilisé pour décrire ce simple langage texte. Et peu importe comment vous développez sur le web, le but de votre serveur est *toujours* de comprendre de simples requêtes composées de texte, et de retourner de simples réponses composées elles aussi de texte.

Symfony2 est construit sur les bases de cette réalité. Que vous le réalisiez ou non, HTTP est quelque chose que vous utilisez tous les jours. Avec Symfony2, vous allez apprendre comment le maîtriser.

Étape 1: Le Client envoie une Requête

Chaque conversation sur le web débute avec une *requête*. La requête est un message textuel créé par un client (par exemple: un navigateur, une application smartphone, etc...) dans un format spécial connu sous le nom d'HTTP. Le client envoie cette requête à un serveur, et puis attend la réponse.

Jetez un oeil à la première partie de l'interaction (la requête) entre un navigateur et le serveur web xkcd:



Dans le langage HTTP, cette requête HTTP ressemblerait à quelque chose comme ça:

Listing 1-1
```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

Ce simple message communique *tout* ce qui est nécessaire concernant la ressource que le client a demandée. La première ligne d'une requête HTTP est la plus importante et contient deux choses: l'URI et la méthode HTTP.

L'URI (par exemple: `/`, `/contact`, etc...) est l'adresse unique ou la localisation qui identifie la ressource que le client veut. La méthode HTTP (par exemple: `GET`) définit ce que vous voulez *faire* avec la ressource.

Les méthodes HTTP sont les *verbes* de la requête et définissent les divers moyens par lesquels vous pouvez agir sur la ressource:

GET	Récupère la ressource depuis le serveur
POST	Crée une ressource sur le serveur
PUT	Met à jour la ressource sur le serveur
DELETE	Supprime la ressource sur le serveur

Avec ceci en mémoire, vous pouvez imaginer ce à quoi ressemblerait une requête HTTP pour supprimer une entrée spécifique d'un blog, par exemple:

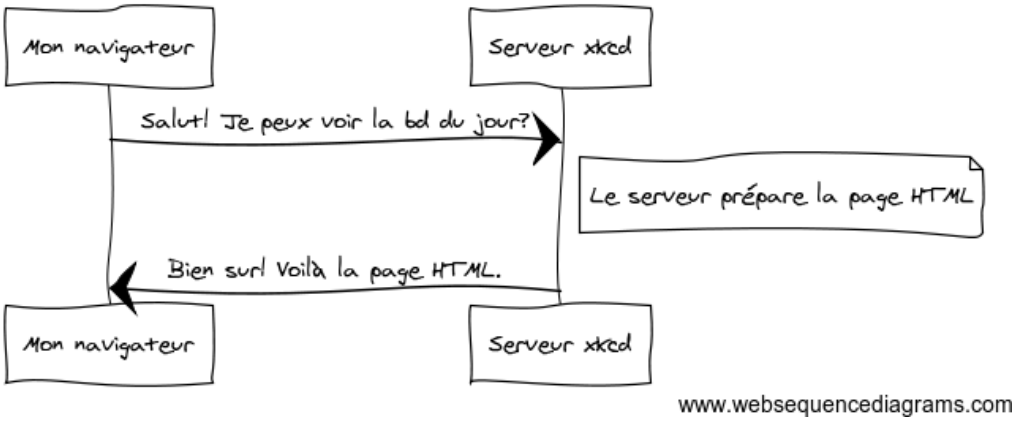
Listing 1-2
```
1 DELETE /blog/15 HTTP/1.1
```

Il y a en fait neuf méthodes HTTP définies par la spécification HTTP, mais beaucoup d'entre elles ne sont pas largement utilisées ou supportées. En réalité, beaucoup de navigateurs modernes ne supportent pas les méthodes `PUT` et `DELETE`.

En plus de la première ligne, une requête HTTP contient invariablement d'autres lignes d'informations appelées entêtes de requête. Les entêtes peuvent fournir un large éventail d'informations telles que l'entête `Host`, le format de réponse que le client accepte (`Accept`) et l'application que le client utilise pour effectuer la requête (`User-Agent`). Beaucoup d'autres entêtes existent et peuvent être trouvés sur la page Wikipedia *List of HTTP header fields*[2] (anglais).

Étape 2: Le Serveur retourne une réponse

Une fois que le serveur a reçu la requête, il connaît exactement quelle ressource le client a besoin (via l'URI) et ce que le client souhaite faire avec cette ressource (via la méthode). Par exemple, dans le cas d'une requête GET, le serveur prépare la ressource et la retourne dans une réponse HTTP. Considérez la réponse du serveur web xkcd :



Traduit en HTTP, la réponse envoyée au navigateur va ressembler à quelque chose comme ça :

Listing 1-3
```
1 HTTP/1.1 200 OK
2 Date: Sat, 02 Apr 2011 21:05:05 GMT
3 Server: lighttpd/1.4.19
```

2. http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

```
Content-Type: text/html

<html>
  <!-- ... code HTML de la BD xkcd -->
</html>
```

La réponse HTTP contient la ressource demandée (le contenu HTML dans ce cas), ainsi que d'autres informations à propos de la réponse. La première ligne est spécialement importante et contient le code de statut de la réponse HTTP (200 dans ce cas). Le code de statut communique le résultat global de la requête retournée au client. A-t-elle réussi ? Y'a-t-il eu une erreur ? Différents codes de statut existent qui indiquent le succès, une erreur, ou que le client a besoin de faire quelque chose (par exemple: rediriger sur une autre page). Une liste complète peut être trouvée sur la page Wikipedia *Liste des codes HTTP*[3] .

Comme la requête, une réponse HTTP contient de l'information additionnelle sous forme d'entêtes HTTP. Par exemple, `Content-Type` est un entête de réponse HTTP très important. Le corps d'une même ressource peut être retournée dans de multiples formats incluant HTML, XML ou JSON et l'entête `Content-Type` utilise les Internet Media Types, comme `text/html`, pour dire au client quel format doit être retourné. Une liste des médias types les plus communs peut être trouvée sur la page Wikipedia *Liste de media type usuels*[4].

De nombreuses autres entêtes existent, dont quelques-uns sont très puissants. Par exemple, certains entêtes peuvent être utilisés pour créer un puissant système de cache.

Requêtes, Réponses et Développement Web

Cette conversation requête-réponse est le procédé fondamental qui régit toute communication sur le web. Et tout aussi important et puissant que ce procédé soit, il est inéluctablement simple.

Le plus important est que : quel que soit le langage que vous utilisez, le type d'application que vous construisez (web, mobile, API JSON), ou la philosophie de développement que vous suivez, l'objectif final d'une application est **toujours** de comprendre chaque requête et de créer et retourner la réponse appropriée.

Symfony est conçu pour correspondre à cette réalité.

Pour en savoir plus à propos de la spécification HTTP, lisez la RFC originale *HTTP 1.1 RFC*[5] ou le *HTTP Bis*[6], qui est un effort actif pour clarifier la spécification originale. Un super outil pour inspecter/vérifier les entêtes de la requête et de la réponse durant votre navigation est l'extension pour Firefox *Live HTTP Headers*[7].

Requêtes et réponses en PHP

Alors comment interagissez-vous avec la «requête» et créez-vous la «réponse» quand vous utilisez PHP ? En réalité, PHP vous abstrait une partie du processus global:

Listing 1-4
```
$uri = $_SERVER['REQUEST_URI'];
$foo = $_GET['foo'];

```

3. http://fr.wikipedia.org/wiki/Liste_des_codes_HTTP
4. http://fr.wikipedia.org/wiki/Type_MIME#Liste_de_media_type_usuels
5. http://www.w3.org/Protocols/rfc2616/rfc2616.html
6. http://datatracker.ietf.org/wg/httpbis/
7. https://addons.mozilla.org/fr/firefox/addon/live-http-headers/

```
4 header('Content-type: text/html');
5 echo 'L\'URI demandée est: '.$uri;
6 echo 'La valeur du paramètre "foo" est: '.$foo;
```

Aussi étrange que cela puisse paraître, cette petite application utilise les informations de la requête HTTP afin de créer une réponse. Plutôt que d'analyser le message texte de la requête HTTP directement, PHP prépare des variables superglobales telles que `$_SERVER` et `$_GET` qui contiennent toute les informations de la requête. De même, au lieu de retourner la réponse texte HTTP formatée, vous pouvez utiliser la fonction `header()` pour créer des entêtes de réponse et simplement délivrer le contenu actuel qui sera la partie contenu du message de la réponse. PHP va ainsi créer une véritable réponse HTTP et la retourner au client :

Listing 1-5

```
1 HTTP/1.1 200 OK
2 Date: Sat, 03 Apr 2011 02:14:33 GMT
3 Server: Apache/2.2.17 (Unix)
4 Content-Type: text/html
5
6 L'URI demandée est: /testing?foo=symfony
7 La valeur du paramètre "foo" est: symfony
```

Requêtes et Réponses dans Symfony

Symfony fournit une alternative à l'approche brute de PHP via deux classes qui vous permettent d'interagir avec la requête et la réponse HTTP de manière plus facile. La classe ***Request***[8] est une simple représentation orientée objet du message de la requête HTTP. Avec elle, vous avez toute l'information de la requête à votre portée:

Listing 1-6

```
1  use Symfony\Component\HttpFoundation\Request;
2
3  $request = Request::createFromGlobals();
4
5  // l'URI demandée (par exemple: /about) sans aucun paramètre
6  $request->getPathInfo();
7
8  // obtenir respectivement des variables GET et POST
9  $request->query->get('foo');
10 $request->request->get('bar', 'valeur par défaut si bar est inexistant');
11
12 // obtenir les variables SERVER
13 $request->server->get('HTTP_HOST');
14
15 // obtenir une instance de UploadedFile identifiée par foo
16 $request->files->get('foo');
17
18 // obtenir la valeur d'un COOKIE value
19 $request->cookies->get('PHPSESSID');
20
21 // obtenir un entête de requête HTTP request header, normalisé en minuscules
22 $request->headers->get('host');
23 $request->headers->get('content_type');
24
```

8. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Request.html

```
$request->getMethod();          // GET, POST, PUT, DELETE, HEAD
$request->getLanguages();       // un tableau des langues que le client accepte
```

En bonus, la classe `Request` effectue beaucoup de travail en arrière-plan dont vous n'aurez jamais à vous soucier. Par exemple, la méthode `isSecure()` vérifie les *trois* valeurs PHP qui peuvent indiquer si oui ou non l'utilisateur est connecté via une connexion sécurisée (c-a-d `https`).

Attributs de ParameterBags et Request

Comme vu ci-dessus, les variables `$_GET` et `$_POST` sont accessibles respectivement via les propriétés publiques `query` et `request`. Chacun de ces objets est un objet *ParameterBag*[9] qui a des méthodes comme *get()*[10], *has()*[11], *all()*[12] et bien d'autres. En fait, chaque propriété publique utilisée dans l'exemple précédent est une instance de ParameterBag.

La classe Request a aussi une propriété publique `attributes` qui contient des données spéciales liées au fonctionnement interne de l'application. Pour le framework Symfony2, la propriété `attributes` contient les valeurs retournées par la route identifiée, comme `_controller`, `id` (si vous utilisez le joker `{id}`), et même le nom de la route (`_route`). La propriété `attributes` existe pour vous permettre d'y stocker des informations spécifiques liées au contexte de la requête.

Symfony fournit aussi une classe `Response` : une simple représentation PHP du message d'une réponse HTTP. Cela permet à votre application d'utiliser une interface orientée objet pour construire la réponse qui doit être retournée au client:

Listing 1-7
```
use Symfony\Component\HttpFoundation\Response;
$response = new Response();

$response->setContent('<html><body><h1>Hello world!</h1></body></html>');
$response->setStatusCode(200);
$response->headers->set('Content-Type', 'text/html');

// affiche les entêtes HTTP suivies du contenu
$response->send();
```

Si Symfony n'offrait rien d'autre, vous devriez néanmoins déjà avoir en votre possession une boîte à outils pour accéder facilement aux informations de la requête et une interface orientée objet pour créer la réponse. Bien que vous appreniez les nombreuses et puissantes fonctions de Symfony, gardez à l'esprit que le but de votre application est toujours *d'interpréter une requête et de créer la réponse appropriée basée sur votre logique applicative*.

Les classes `Request` et `Response` font partie d'un composant autonome inclus dans Symfony appelé `HttpFoundation`. Ce composant peut être utilisé de manière entièrement indépendante de Symfony et fournit aussi des classes pour gérer les sessions et les uploads de fichier.

9. http://api.symfony.com/master/Symfony/Component/HttpFoundation/ParameterBag.html
10. http://api.symfony.com/master/Symfony/Component/HttpFoundation/ParameterBag.html#get()
11. http://api.symfony.com/master/Symfony/Component/HttpFoundation/ParameterBag.html#has()
12. http://api.symfony.com/master/Symfony/Component/HttpFoundation/ParameterBag.html#all()

Le Parcours de la Requête à la Réponse

Comme HTTP lui-même, les objets `Request` et `Response` sont assez simples. La partie difficile de la création d'une application est d'écrire ce qui vient entre les deux. En d'autres termes, le réel travail commence lors de l'écriture du code qui interprète l'information de la requête et crée la réponse.

Votre application fait probablement beaucoup de choses comme envoyer des emails, gérer des soumissions de formulaires, enregistrer des choses dans une base de données, délivrer des pages HTML et protéger du contenu de façon sécurisée. Comment pouvez-vous vous occuper de tout cela tout en conservant votre code organisé et maintenable ?

Symfony a été créé pour résoudre ces problématiques afin que vous n'ayez pas à le faire vous-même.

Le Contrôleur Frontal

Traditionnellement, les applications étaient construites de telle sorte que chaque « page » d'un site avait son propre fichier physique:

Listing 1-8
```
index.php
contact.php
blog.php
```

Il y a plusieurs problèmes avec cette approche, ce qui inclut la non-flexibilité des URLs (que se passait-il si vous souhaitiez changer `blog.php` en `news.php` sans que tous vos liens existants ne cessent de fonctionner ?) et le fait que chaque fichier *doive* manuellement inclure tout un ensemble de fichiers coeurs pour que la sécurité, les connexions à la base de données et le « look » du site puissent rester cohérents.

Une bien meilleure solution est d'utiliser un simple fichier PHP appelé *contrôleur frontal*: qui s'occupe de chaque requête arrivant dans votre application. Par exemple:

`/index.php`	exécute `index.php`
`/index.php/contact`	exécute `index.php`
`/index.php/blog`	exécute `index.php`

En utilisant la fonction `mod_rewrite` d'Apache (ou son équivalent avec d'autres serveurs web), les URLs peuvent être facilement réécrites afin de devenir simplement `/`, `/contact` et `/blog`.

Maintenant, chaque requête est gérée exactement de la même façon. Plutôt que d'avoir des URLs individuelles exécutant des fichiers PHP différents, le contrôleur frontal est *toujours* exécuté, et le routage (« routing ») des différentes URLs vers différentes parties de votre application est effectué en interne. Cela résoud les deux problèmes de l'approche originale. Presque toutes les applications web modernes font ça - incluant les applications comme WordPress.

Rester Organisé

Dans votre contrôleur frontal, vous devez déterminer quelle portion de code doit être exécuté et quel est le contenu qui doit être retourné. Vous le savoir, vous allez devoir inspecter l'URI entrante et exécuter les différentes parties de votre code selon cette valeur. Cela peut rapidement devenir moche:

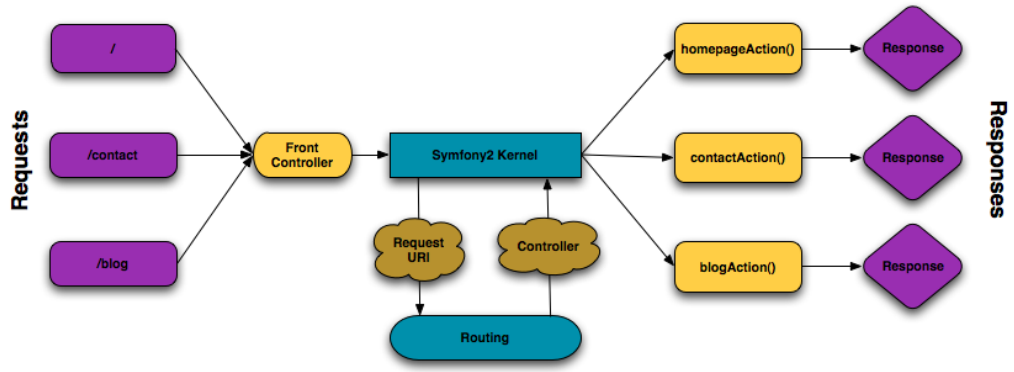
Listing 1-9
```
// index.php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
```

```php
$request = Request::createFromGlobals();
$path = $request->getPathInfo(); // Le chemin de l'URI demandée

if (in_array($path, array('', '/'))) {
    $response = new Response('Bienvenue sur le site.');
} elseif ($path == '/contact') {
    $response = new Response('Contactez nous');
} else {
    $response = new Response('Page non trouvée.', 404);
}
$response->send();
```

Résoudre ce problème peut être difficile. Heureusement, c'est *exactement* ce pourquoi Symfony a été conçu.

Le Déroulement d'une Application Symfony

Quand vous laissez Symfony gérer chaque requête, la vie est beaucoup plus facile. Symfony suit un schéma simple et identique pour toutes les requêtes :



Les requêtes entrantes sont interprétées par le routing et passées aux fonctions des contrôleurs qui retournent des objets `Response`.

Chaque « page » de votre site est définie dans un fichier de configuration du routing qui relie différentes URLs à différentes fonctions PHP. Le travail de chaque fonction PHP, appelée *contrôleur*, est de créer puis retourner un objet `Response`, construit à partir des informations de la requête, à l'aide des outils mis à disposition par le framework. En d'autres termes, le contrôleur est le lieu où *votre* code se trouve : c'est là que vous interprétez la requête et que vous créez une réponse.

C'est si facile ! Revoyons cela :

- Chaque requête exécute un même et unique fichier ayant le rôle de contrôleur frontal;
- Le système de routing détermine quelle fonction PHP doit être exécutée en se basant sur les informations provenant de la requête et la configuration de routage que vous avez créé;
- La fonction PHP correcte est exécutée, là où votre code crée et retourne l'objet `Response` approprié.

Une Requête Symfony en Action

Sans aller trop loin dans les détails, voyons ce procédé en action. Supposons que vous vouliez ajouter une page `/contact` à votre application Symfony. Premièrement, commencez par ajouter une entrée pour `/contact` dans votre fichier de configuration du routing:

Listing 1-10
```yaml
# app/config/routing.yml
contact:
    path:     /contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
```

Cet exemple utilise *YAML* pour définir la configuration de routage. Cette dernière peut aussi être écrite dans d'autres formats comme XML ou PHP.

Lorsque quelqu'un visite la page `/contact`, il y a correspondance avec cette route, et le contrôleur spécifié est exécuté. Comme vous l'apprendrez dans le *chapitre sur le routage*, la chaîne de caractères `AcmeDemoBundle:Main:contact` est une syntaxe raccourcie qui pointe vers une méthode PHP spécifique `contactAction` dans la classe appelée `MainController`:

Listing 1-11
```php
// src/Acme/DemoBundle/Controller/MainController.php
namespace Acme\DemoBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class MainController
{
    public function contactAction()
    {
        return new Response('<h1>Contactez nous!</h1>');
    }
}
```

Dans cet exemple très simple, le contrôleur crée simplement un objet *Response*[13] contenant l'HTML "<h1>Contactez nous!</h1>". Dans le *chapitre Contrôleur*, vous allez apprendre comment un contrôleur peut retourner des templates, permettant à votre code de « présentation » (c-a-d du code qui retourne du HTML) de se trouver dans un fichier de template séparé. Cela libère le contrôleur et lui permet de s'occuper seulement des choses complexes : interagir avec la base de données, gérer les données soumises, ou envoyer des emails.

Symfony2: Construisez votre application, pas vos outils

Vous savez maintenant que le but de toute application est d'interpréter chaque requête entrante et de créer une réponse appropriée. Avec le temps, une application grandit et il devient plus difficile de garder le code organisé et maintenable. Invariablement, les mêmes tâches complexes reviennent encore et toujours : persister des éléments dans la base de données, afficher et réutiliser des templates, gérer des soumissions de formulaires, envoyer des emails, valider des entrées d'utilisateurs et gérer la sécurité.

La bonne nouvelle est qu'aucun de ces problèmes n'est unique. Symfony fournit un framework rempli d'outils qui vous permettent de construire votre application, mais pas vos outils. Avec Symfony2, rien ne vous est imposé : vous êtes libre d'utiliser le framework Symfony en entier, ou juste une partie de Symfony indépendamment.

Outils Autonomes: Les *Composants* Symfony2

Donc *qu'est-ce* que Symfony2? Premièrement, Symfony2 est une collection de plus de vingt bibliothèques indépendantes qui peuvent être utilisées dans *n'importe quel* projet PHP. Ces bibliothèques, appelées les

13. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html

Composants Symfony2, contiennent des choses utiles en toute situation, quelle que soit la manière dont votre projet est développé. Pour en nommer quelques-unes :

- *HttpFoundation* - Contient les classes `Request` et `Response`, ainsi que d'autres classes pour la gestion des sessions et des uploads de fichiers;
- *Routing* - Un puissant et rapide système qui vous permet de lier une URI spécifique (par exemple: `/contact`) à l'information lui permettant de savoir comment gérer cette requête (par exemple: exécute la méthode `contactAction()`);
- *Form*[14] - Un framework complet et flexible pour la création de formulaires et la gestion de la soumission de ces derniers;
- *Validator*[15] Un système permettant de créer des règles à propos de données et de valider ou non les données utilisateurs soumises suivant ces règles;
- *ClassLoader* Une bibliothèque pour le chargement automatique (« autoloading ») qui permet aux classes PHP d'être utilisées sans avoir besoin d'`inclure` (« require ») manuellement les fichiers contenant ces dernières;
- *Templating* Une boîte à outils pour afficher des templates, gérer leur héritage (c-a-d qu'un template est décoré par un layout) et effectuer d'autres tâches communes aux templates;
- *Security*[16] - Une puissante bibliothèque pour gérer tous les types de sécurité dans une application;
- *Translation*[17] Un framework pour traduire les chaînes de caractères dans votre application.

Chacun de ces composants est découplé et peut être utilisé dans *n'importe quel* projet PHP, que vous utilisiez le framework Symfony2 ou non. Chaque partie est faite pour être utilisée si besoin est, et remplacée quand cela est nécessaire.

La Solution Complète: Le *Framework* Symfony2

Donc finalement, *qu'est-ce* que le *Framework* Symfony2 ? Le *Framework Symfony2* est une bibliothèque PHP qui accomplit deux tâches distinctes :

1. Fournir une sélection de composants (les Composants Symfony2) et des bibliothèques tierces (ex *Swiftmailer*[18] pour envoyer des emails);
2. Fournir une configuration et une bibliothèque « colle » qui lie toutes ces pièces ensembles.

Le but du framework est d'intégrer plein d'outils indépendants afin de fournir une expérience substantielle au développeur. Même le framework lui-même est un bundle Symfony2 (c-a-d un plugin) qui peut être configuré ou remplacé entièrement.

Symfony2 fournit un puissant ensemble d'outils pour développer rapidement des applications web sans pour autant s'imposer à votre application. Les utilisateurs normaux peuvent commencer rapidement à développer en utilisant une distribution Symfony2, ce qui fournit un squelette de projet avec des paramètres par défaut. Pour les utilisateurs avancés, le ciel est la seule limite.

14. https://github.com/symfony/Form
15. https://github.com/symfony/Validator
16. https://github.com/symfony/Security
17. https://github.com/symfony/Translation
18. http://swiftmailer.org/

Chapter 2
Symfony2 versus PHP pur

Pourquoi est-ce que Symfony2 est mieux que simplement ouvrir un fichier et écrire du PHP pur?

Si vous n'avez jamais utilisé un framework PHP, que vous n'êtes pas familier avec la philosophie MVC, ou simplement que vous vous demandez pourquoi il y a tant d'*agitation* autour de Symfony2, ce chapitre est pour vous. Au lieu de vous *dire* que Symfony2 vous permet de développer plus rapidement de meilleures applications qu'en PHP pur, vous allez le voir de vos propres yeux.

Dans ce chapitre, vous allez écrire une application simple en PHP, puis refactoriser le code afin d'en améliorer l'organisation. Vous voyagerez dans le temps, en comprenant les décisions qui ont fait évoluer le développement web au cours de ces dernières années.

D'ici la fin, vous verrez comment Symfony peut vous épargner les tâches banales et vous permet de reprendre le contrôle de votre code.

Un simple blog en PHP

Dans ce chapitre, vous bâtirez une application de blog en utilisant uniquement du code PHP. Pour commencer, créez une page qui affiche les billets du blog qui ont été sauvegardés dans la base de données. Écrire en pur PHP est « rapide et sale » :

Listing 2-1

```
<?php
// index.php
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);
?>

<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
```

```
16          <ul>
17              <?php while ($row = mysql_fetch_assoc($result)): ?>
18              <li>
19                  <a href="/show.php?id=<?php echo $row['id'] ?>">
20                      <?php echo $row['title'] ?>
21                  </a>
22              </li>
23              <?php endwhile; ?>
24          </ul>
25      </body>
26  </html>
27
28  <?php
29  mysql_close($link);
30  ?>
```

C'est rapide à écrire, rapide à exécuter et, comme votre application évoluera, impossible à maintenir. Il y a plusieurs problèmes qui doivent être résolus :

- **aucune gestion d'erreur** : Que se passe-t-il si la connexion à la base de données échoue?
- **mauvaise organisation** : si votre application évolue, ce fichier deviendra de moins en moins maintenable. Où devrez-vous mettre le code qui traite la soumission d'un formulaire? Comment pourrez-vous valider les données? Où devrez-vous mettre le code qui envoie des courriels?
- **Difficulté de réutiliser du code** : puisque tout se trouve dans un seul fichier, il est impossible de réutiliser des parties de l'application pour d'autres pages du blog.

Un autre problème non mentionné ici est le fait que la base de données est liée à MySQL. Même si le sujet n'est pas couvert ici, Symfony intègre *Doctrine*[1], une bibliothèque dédiée à l'abstraction des base de données et au mapping objet-relationnel.

Retroussons-nous les manches et résolvons ces problèmes ainsi que d'autres.

Isoler la présentation

Le code sera mieux structuré en séparant la logique « applicative » du code qui s'occupe de la « présentation » HTML :

Listing 2-2

```
1  <?php
2  // index.php
3  $link = mysql_connect('localhost', 'myuser', 'mypassword');
4  mysql_select_db('blog_db', $link);
5
6  $result = mysql_query('SELECT id, title FROM post', $link);
7
8  $posts = array();
9  while ($row = mysql_fetch_assoc($result)) {
10     $posts[] = $row;
11 }
12
13 mysql_close($link);
14
```

1. http://www.doctrine-project.org

```
15 // inclut le code de la présentation HTML
16 require 'templates/list.php';
```

Le code HTML est maintenant dans un fichier séparé (`templates/list.php`), qui est essentiellement un fichier HTML qui utilise une syntaxe PHP de template :

Listing 2-3

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>List of Posts</title>
5      </head>
6      <body>
7          <h1>List of Posts</h1>
8          <ul>
9              <?php foreach ($posts as $post): ?>
10             <li>
11                 <a href="/read?id=<?php echo $post['id'] ?>">
12                     <?php echo $post['title'] ?>
13                 </a>
14             </li>
15             <?php endforeach; ?>
16         </ul>
17     </body>
18 </html>
```

Par convention, le fichier qui contient la logique applicative (`index.php`) est appelé « contrôleur ». Le terme *contrôleur* est un mot que vous allez entendre souvent, quel que soit le langage ou le framework utilisé. Il fait simplement référence à *votre* code qui traite les entrées de l'utilisateur et prépare une réponse.

Dans notre cas, le contrôleur prépare les données de la base de données et inclut ensuite un template pour présenter ces données. Comme le contrôleur est isolé, vous pouvez facilement changer *uniquement* le fichier de template si vous désirez afficher les billets du blog dans un autre format (par exemple `list.json.php` pour un format JSON).

Isoler la logique applicative

Pour l'instant, l'application ne contient qu'une seule page. Mais que faire si une deuxième page a besoin d'utiliser la même connexion à la base de données, ou le même tableau de billets du blog? Refactorisez le code pour que le comportement principal et les fonctions d'accès aux données de l'application soient isolés dans un nouveau fichier appelé `model.php` :

Listing 2-4

```
1  <?php
2  // model.php
3  function open_database_connection()
4  {
5      $link = mysql_connect('localhost', 'myuser', 'mypassword');
6      mysql_select_db('blog_db', $link);
7
8      return $link;
9  }
10
11 function close_database_connection($link)
12 {
13     mysql_close($link);
14 }
```

```php

function get_all_posts()
{
    $link = open_database_connection();

    $result = mysql_query('SELECT id, title FROM post', $link);
    $posts = array();
    while ($row = mysql_fetch_assoc($result)) {
        $posts[] = $row;
    }
    close_database_connection($link);

    return $posts;
}
```

Le nom du fichier `model.php` est utilisé, car la logique et l'accès aux données de l'application sont traditionnellement connus sous le nom de couche « modèle ». Dans une application bien structurée, la majorité du code représentant la logique métier devrait se trouver dans le modèle (plutôt que dans le contrôleur). Et contrairement à cet exemple, seulement une portion (ou aucune) du modèle est en fait concernée par l'accès à la base de données.

Le contrôleur (`index.php`) est maintenant très simple :

Listing 2-5
```php
<?php
require_once 'model.php';

$posts = get_all_posts();

require 'templates/list.php';
```

Maintenant, la seule responsabilité du contrôleur est de récupérer les données de la couche modèle de l'application (le modèle) et d'appeler le template à afficher ces données. C'est un exemple très simple du patron de conception Modèle-Vue-Contrôleur.

Isoler le layout

À ce point-ci, l'application a été refactorisée en trois parties distinctes, offrant plusieurs avantages et l'opportunité de réutiliser pratiquement la totalité du code pour d'autres pages.

La seule partie du code qui *ne peut pas* être réutilisée est le layout de la page. Corrigez cela en créant un nouveau fichier `layout.php` :

Listing 2-6
```php
<!-- templates/layout.php -->
<!DOCTYPE html>
<html>
    <head>
        <title><?php echo $title ?></title>
    </head>
    <body>
        <?php echo $content ?>
    </body>
</html>
```

Le template (`templates/list.php`) peut maintenant simplement « hériter » du layout :

Listing 2-7
```php
<?php $title = 'List of Posts' ?>

<?php ob_start() ?>
    <h1>List of Posts</h1>
    <ul>
        <?php foreach ($posts as $post): ?>
        <li>
            <a href="/read?id=<?php echo $post['id'] ?>">
                <?php echo $post['title'] ?>
            </a>
        </li>
        <?php endforeach; ?>
    </ul>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Vous avez maintenant une méthode qui permet la réutilisation du layout. Malheureusement, pour accomplir cela, vous êtes forcé d'utiliser des fonctions PHP (`ob_start()`, `ob_get_clean()`) dans le template. Symfony utilise un composant de `Templates` qui permet d'accomplir ce résultat proprement et facilement. Vous le verrez en action bientôt.

Ajout d'une page pour afficher un billet

La page « liste » a maintenant été refactorisée afin que le code soit mieux organisé et réutilisable. Pour le prouver, ajoutez une page qui va afficher un billet identifié par un paramètre de requête `id`.

Pour commencer, créez une fonction dans le fichier `model.php` qui récupère un seul billet du blog en fonction d'un id passé en paramètre:

Listing 2-8
```php
// model.php
function get_post_by_id($id)
{
    $link = open_database_connection();

    $id = intval($id);
    $query = 'SELECT date, title, body FROM post WHERE id = '.$id;
    $result = mysql_query($query);
    $row = mysql_fetch_assoc($result);

    close_database_connection($link);

    return $row;
}
```

Puis créez un nouveau fichier appelé `show.php` - le contrôleur pour cette nouvelle page :

Listing 2-9
```php
<?php
require_once 'model.php';

$post = get_post_by_id($_GET['id']);

require 'templates/show.php';
```

Finalement, créez un nouveau fichier de template - `templates/show.php` - afin d'afficher un billet du blog :

Listing 2-10
```php
<?php $title = $post['title'] ?>

<?php ob_start() ?>
    <h1><?php echo $post['title'] ?></h1>

    <div class="date"><?php echo $post['date'] ?></div>
    <div class="body">
        <?php echo $post['body'] ?>
    </div>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Créer cette nouvelle page est vraiment facile et aucun code n'est dupliqué. Malgré tout, cette page introduit des problèmes persistants qu'un framework peut résoudre. Par exemple, un paramètre de requête `id` manquant ou invalide va générer une erreur fatale. Il serait préférable de générer une erreur 404, mais cela ne peut pas vraiment être fait facilement. Pire, comme vous avez oublié de nettoyer le paramètre `id` avec la fonction `intval()`, votre base de données est sujette à des attaques d'injection de SQL.

Un autre problème est que chaque fichier contrôleur doit inclure le fichier `model.php`. Que se passerait-il si chaque contrôleur devait soudainement inclure un fichier additionnel ou effectuer une quelconque tâche globale (comme renforcer la sécurité)? Dans l'état actuel, tout nouveau code devra être ajouté à chaque fichier contrôleur. Si vous oubliez de modifier un fichier, il serait bon que ce ne soit pas relié à la sécurité...

Un « contrôleur frontal » à la rescousse

La solution est d'utiliser un *contrôleur frontal* (front controller): un fichier PHP à travers lequel chaque requête sera traitée. Avec un contrôleur frontal, les URIs de l'application changent un peu, mais elles sont plus flexibles :

Listing 2-11
```
Sans contrôleur frontal
/index.php          => page de liste des billets (exécution de index.php)
/show.php           => page d'affichage d'un billet (exécution de show.php)

avec index.php comme contrôleur frontal
/index.php          => page de liste des billets (exécution de index.php)
/index.php/show     => page d'affichage d'un billet (exécution de index.php)
```

La portion `index.php` de l'URI peut être enlevée en utilisant les règles de réécriture d'URI d'Apache (ou équivalent). Dans ce cas, l'URI de la page de détail d'un billet serait simplement `/show`.

En utilisant un contrôleur frontal, un seul fichier PHP (`index.php` dans notre cas) traite *chaque* requête. Pour la page de détail d'un billet, `/index.php/show` va donc exécuter le fichier `index.php`, qui est maintenant responsable de router en interne les requêtes en fonction de l'URI complète. Comme vous allez le voir, un contrôleur frontal est un outil très puissant.

Créer le contrôleur frontal

Vous êtes sur le point de franchir une étape *importante* de l'application. Avec un seul fichier qui traite toutes les requêtes, vous pouvez centraliser des fonctionnalités comme la gestion de la sécurité, le chargement de la configuration, et le routage. Dans cette application, `index.php` doit être assez intelligent pour afficher la page de liste des billets *ou* la page de détail d'un billet en fonction de l'URI demandée :

Listing 2-12
```php
<?php
// index.php

// charge et initialise les bibliothèques globales
require_once 'model.php';
require_once 'controllers.php';

// route la requête en interne
$uri = $_SERVER['REQUEST_URI'];
if ('/index.php' == $uri) {
    list_action();
} elseif ('/index.php/show' == $uri && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Page Not Found</h1></body></html>';
}
```

Pour des raisons d'organisation, les contrôleurs (initialement `index.php` et `show.php`) sont maintenant des fonctions PHP et ont été placés dans le fichier `controllers.php` :

Listing 2-13
```php
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

En tant que contrôleur frontal, `index.php` assume un nouveau rôle, celui d'inclure les bibliothèques principales et de router l'application pour que l'un des deux contrôleurs (les fonctions `list_action()` et `show_action()`) soit appelé. En réalité, le contrôleur frontal commence à ressembler et à agir comme le mécanisme de Symfony2 qui prend en charge et achemine les requêtes.

Un autre avantage du contrôleur frontal est d'avoir des URIs flexibles. Veuillez noter que l'URL de la page d'affichage d'un billet peut être changée de `/show` à `/read` en changeant le code à un seul endroit. Sans le contrôleur frontal, il aurait fallu renommer un fichier. Avec Symfony2, les URLs sont encore plus flexibles.

Jusqu'ici, l'application est passée d'un seul fichier PHP à une organisation qui permet la réutilisation du code. Vous devriez être plus heureux, mais loin d'être satisfait. Par exemple, le système de routing est inconstant, et ne reconnaîtrait pas que la page de liste de billets (`/index.php`) devrait aussi être accessible via `/` (si Apache rewrite est activé). Aussi, au lieu de développer une application de blog, beaucoup de temps a été consacré à l'« architecture » du code (par exemple le routing, l'appel aux contrôleurs, aux

templates...). Plus de temps devrait être consacré à la prise en charge des formulaires, la validation des champs, la journalisation et la sécurité. Pourquoi réinventer des solutions pour tout ces problèmes ?

Ajoutez une touche de Symfony2

Symfony2 à la rescousse. Avant d'utiliser Symfony2, vous devez d'abord le télécharger. Vous pouvez le faire en utilisant Composer, qui se chargera de télécharger la bonne version ainsi que toutes ses dépendances, et qui chargera également les classes de manière automatique (grâce à un autoloader). Un autoloader est un outil qui rend possible l'utilisation de classes sans avoir à inclure les fichiers qui contiennent chaque classe.

Dans votre répertoire racine, créez un fichier `composer.json` avec le contenu suivant :

Listing 2-14
```
{
    "require": {
        "symfony/symfony": "2.3.*"
    },
    "autoload": {
        "files": ["model.php","controllers.php"]
    }
}
```

Ensuite, *téléchargez Composer*[2] puis exécutez la commande suivante, qui téléchargera Symfony dans un répertoire /vendor :

Listing 2-15
```
$ php composer.phar install
```

En plus de télécharger vos dépendances, Composer génère un fichier `vendor/autoload.php`, qui prendra en charge l'autoloading pour tous les fichiers du framework Symfony, de même que les fichiers spécifiés dans la section autoload de votre `composer.json`.

La philosophie de base de Symfony est que la principale activité d'une application est d'interpréter chaque requête et de retourner une réponse. Pour cela, Symfony2 fournit les classes *`Request`*[3] et *`Response`*[4]. Ces classes sont des représentations orientées-objet des requêtes HTTP brutes qui sont en train d'être exécutées et d'une réponse HTTP qui est retournée. Utilisez-les pour améliorer le blog :

Listing 2-16
```
<?php
// index.php
require_once 'vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();

$uri = $request->getPathInfo();
if ('/' == $uri) {
    $response = list_action();
} elseif ('/show' == $uri && $request->query->has('id')) {
    $response = show_action($request->query->get('id'));
} else {
    $html = '<html><body><h1>Page Not Found</h1></body></html>';
    $response = new Response($html, 404);
}
```

2. http://getcomposer.org/download/
3. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Request.html
4. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html

```
19
20 // affiche les entêtes et envoie la réponse
21 $response->send();
```

Les contrôleurs sont maintenant chargés de retourner un objet `Response`. Pour simplifier les choses, vous pouvez ajouter une nouvelle fonction `render_template()`, qui agit un peu comme le moteur de template de Symfony2 :

Listing 2-17

```
1  // controllers.php
2  use Symfony\Component\HttpFoundation\Response;
3
4  function list_action()
5  {
6      $posts = get_all_posts();
7      $html = render_template('templates/list.php', array('posts' => $posts));
8
9      return new Response($html);
10 }
11
12 function show_action($id)
13 {
14     $post = get_post_by_id($id);
15     $html = render_template('templates/show.php', array('post' => $post));
16
17     return new Response($html);
18 }
19
20 // fonction helper pour afficher un template
21 function render_template($path, array $args)
22 {
23     extract($args);
24     ob_start();
25     require $path;
26     $html = ob_get_clean();
27
28     return $html;
29 }
```

En intégrant une petite partie de Symfony2, l'application est plus flexible et fiable. La requête (`Request`) permet d'accéder aux informations d'une requête HTTP. De manière plus spécifique, la méthode `getPathInfo()` retourne une URI épurée (retourne toujours `/show` et jamais `/index.php/show`). Donc, même si l'utilisateur va à `/index.php/show`, l'application est assez intelligente pour router la requête vers `show_action()`.

L'objet `Response` offre de la flexibilité pour construire une réponse HTTP, permettant d'ajouter des entêtes HTTP et du contenu au travers d'une interface orientée objet. Et même si les réponses de cette application sont simples, cette flexibilité sera un atout lorsque l'application évoluera.

L'application exemple en Symfony2

Le blog a *beaucoup* évolué, mais il contient beaucoup de code pour une si simple application. Durant cette évolution, vous avez créé un mécanisme simple de routage et une méthode utilisant `ob_start()` et `ob_get_clean()` pour afficher les templates. Si, pour une raison, vous deviez continuer à construire ce

« framework », vous pourriez utiliser les composants indépendants *Routing*[5] et *Templating*[6] de Symfony, qui apportent déjà une solution à ces problèmes.

Au lieu de résoudre à nouveau des problèmes classiques, vous pouvez laisser Symfony2 s'en occuper pour vous. Voici la même application, en utilisant cette fois-ci Symfony2:

Listing 2-18
```php
// src/Acme/BlogBundle/Controller/BlogController.php
namespace Acme\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function listAction()
    {
        $posts = $this->get('doctrine')->getManager()
            ->createQuery('SELECT p FROM AcmeBlogBundle:Post p')
            ->execute();

        return $this->render(
            'AcmeBlogBundle:Blog:list.html.php',
            array('posts' => $posts)
        );
    }

    public function showAction($id)
    {
        $post = $this->get('doctrine')
            ->getManager()
            ->getRepository('AcmeBlogBundle:Post')
            ->find($id)
        ;

        if (!$post) {
            // cause the 404 page not found to be displayed
            throw $this->createNotFoundException();
        }

        return $this->render(
            'AcmeBlogBundle:Blog:show.html.php',
            array('post' => $post)
        );
    }
}
```

Les deux contrôleurs sont toujours simples. Chacun utilise la *librairie ORM Doctrine* pour récupérer les objets de la base de données et le composant de `Templating` pour afficher les templates et retourner un objet `Response`. Le template qui affiche la liste est maintenant un peu plus simple :

Listing 2-19
```php
<!-- src/Acme/BlogBundle/Resources/views/Blog/list.html.php -->
<?php $view->extend('::layout.html.php') ?>

<?php $view['slots']->set('title', 'List of Posts') ?>

<h1>List of Posts</h1>
<ul>
```

5. https://github.com/symfony/Routing
6. https://github.com/symfony/Templating

```
    <?php foreach ($posts as $post): ?>
    <li>
        <a href="<?php echo $view['router']->generate(
            'blog_show',
            array('id' => $post->getId())
        ) ?>">
            <?php echo $post->getTitle() ?>
        </a>
    </li>
    <?php endforeach; ?>
</ul>
```

Le layout est à peu près identique :

Listing 2-20
```
<!-- app/Resources/views/layout.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <title><?php echo $view['slots']->output(
            'title',
            'Default title'
        ) ?></title>
    </head>
    <body>
        <?php echo $view['slots']->output('_content') ?>
    </body>
</html>
```

Le template d'affichage d'un billet est laissé comme exercice, cela devrait être assez simple si vous vous basez sur le template de liste.

Lorsque le moteur de Symfony2 (appelé `Kernel`) démarre, il a besoin d'une table qui définit quels contrôleurs exécuter en fonction des informations de la requête. Une table de routage fournit cette information dans un format lisible :

Listing 2-21
```
# app/config/routing.yml
blog_list:
    path:     /blog
    defaults: { _controller: AcmeBlogBundle:Blog:list }

blog_show:
    path:     /blog/show/{id}
    defaults: { _controller: AcmeBlogBundle:Blog:show }
```

Maintenant que Symfony2 prend en charge toutes les taches banales, le contrôleur frontal est extrêmement simple. Et comme il fait très peu de chose, vous n'aurez jamais à le modifier une fois que vous l'aurez créé (et si vous utilisez une distribution de Symfony2, vous n'aurez même pas à le créer !):

Listing 2-22
```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;
```

```
6
7  $kernel = new AppKernel('prod', false);
8  $kernel->handle(Request::createFromGlobals())->send();
```

Le seul rôle du contrôleur frontal est d'initialiser le moteur Symfony2 (`Kernel`) et de lui passer à un objet `Request` à traiter. Le coeur de Symfony2 utilise alors la table de routage pour déterminer quel contrôleur appeler. Comme précédemment, c'est à la méthode du contrôleur de retourner un objet `Response`.

Pour une représentation visuelle qui montre comment Symfony2 traite chaque requête, voir *le diagramme de flux de contrôle d'une requête*.

En quoi Symfony2 tient ses promesses

Dans les chapitres suivants, vous en apprendrez plus sur le fonctionnement chaque élément de Symfony et sur la structure recommandée d'un projet. Pour l'instant, voyons en quoi la migration du blog depuis une version PHP en une version Symfony2 nous simplifie la vie :

- Votre application est constituée de **code clair et bien organisé** (même si Symfony ne vous force pas à le faire). Cela encourage la **réutilisabilité** et permet aux nouveaux développeurs d'être productifs sur votre projet plus rapidement;
- 100% du code que vous écrivez est pour *votre* application. Vous **n'avez pas à développer ou à maintenir des outils de bas niveau** tel que *le chargeur automatique*, *le routage*, ou *les contrôleurs*;
- Symfony2 vous donne **accès à des outils open source** comme Doctrine et le composants de templates, de sécurité, de formulaires, de validation et de traduction (pour n'en nommer que quelques-uns);
- L'application profite maintenant d'**URLs complètement flexibles**, grâce au composant de routage (`Routing`);
- L'architecture centrée autour du protocole HTTP vous donne accès à des outils puissants tels que le **cache HTTP** permis par le **cache HTTP interne de Symfony2** ou par d'autres outils plus puissants tels que *Varnish*[7]. Ce point est couvert dans un chapitre consacré au *cache*.

Et peut-être le point le plus important, en utilisant Symfony, vous avez maintenant accès à un ensemble d'**outils de qualité open source développés par la communauté Symfony2** ! Un large choix d'outils de la communauté Symfony2 se trouve sur *KnpBundles.com*[8].

De meilleurs templates

Si vous choisissez de l'utiliser, Symfony2 vient de facto avec un moteur de template appelé *Twig*[9] qui rend les templates plus rapides à écrire et plus faciles à lire. Cela veut dire que l'application exemple pourrait contenir moins de code ! Prenez par exemple, le template de liste de billets écrit avec Twig :

Listing 2-23
```
1  {# src/Acme/BlogBundle/Resources/views/Blog/list.html.twig #}
2  {% extends "::layout.html.twig" %}
3
4  {% block title %}List of Posts{% endblock %}
5
6  {% block body %}
7      <h1>List of Posts</h1>
8      <ul>
```

7. https://www.varnish-cache.org
8. http://knpbundles.com/
9. http://twig.sensiolabs.org

```
        {% for post in posts %}
        <li>
            <a href="{{ path('blog_show', {'id': post.id}) }}">
                {{ post.title }}
            </a>
        </li>
        {% endfor %}
    </ul>
{% endblock %}
```

Le template du layout associé `layout.html.twig` est encore plus simple à écrire :

Listing 2-24

```
{# app/Resources/views/layout.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <title>{% block title %}Default title{% endblock %}</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Twig est très bien supporté par Symfony2. Et bien que les templates PHP vont toujours être supportés par Symfony2, nous allons continuer à vanter les nombreux avantages de Twig. Pour plus d'information, voir le *chapitre sur les templates*.

Apprenez en lisant le Cookbook

- *Comment utiliser PHP plutôt que Twig dans les templates*
- *Comment définir des contrôleurs en tant que Services*

Chapter 3
Installer et Configurer Symfony

Le but de ce chapitre est de vous permettre de démarrer avec une application construite avec Symfony. Heureusement, Symfony propose un système de « distributions ». Ce sont des projets Symfony fonctionnels « pour démarrer » que vous pouvez télécharger et qui vous permettent de développer immédiatement.

Si vous cherchez des instructions sur la meilleure façon de créer un nouveau projet et de le stocker dans un gestionnaire de versions, lisez Utiliser un Gestionnaire de Versions.

Télécharger une Distribution Symfony2

Premièrement, vérifiez que vous avez installé et configuré un serveur web (comme Apache) avec PHP 5.3.8 ou supérieur. Pour plus d'informations sur les prérequis Symfony2, lisez le chapitre *prérequis*.

Les « distributions » Symfony2 sont des applications entièrement fonctionnelles qui incluent les bibliothèques du coeur de Symfony2, une sélection de bundles utiles, une arborescence pratique et une configuration par défaut. Quand vous téléchargez une distribution Symfony2, vous téléchargez un squelette d'application qui peut être immédiatement utilisé pour commencer à développer votre application.

Commencez par visiter la page de téléchargement de Symfony2 à *http://symfony.com/download*[1]. Sur cette page, vous verrez la *Symfony Standard Edition*, qui est la principale distribution Symfony2. Il y a deux façons de démarrer votre projet :

1. `http://symfony.com/download`

Option 1) « Composer »

Composer[2] est une bibliothèque de gestion de dépendances pour PHP, que vous pouvez utiliser pour télécharger l'Édition Standard de Symfony2.

Commencez par *télécharger Composer*[3] dans un quelconque répertoire sur votre ordinateur. Si vous avez curl d'installé, cela est aussi facile que ce qui suit :

Listing 3-1
```
1 curl -s https://getcomposer.org/installer | php
```

Si votre ordinateur n'est pas prêt à utiliser Composer, vous allez voir quelques recommandations lors de l'exécution de cette commande. Suivez ces recommandations afin que Composer puisse fonctionner correctement sur votre machine locale.

« Composer » est un fichier PHAR exécutable, que vous pouvez utiliser pour télécharger la Distribution Standard :

Listing 3-2
```
1 $ php composer.phar create-project symfony/framework-standard-edition /path/to/webroot/
  Symfony 2.3.0
```

Pour une version exacte, remplacez *2.3.0* par la dernière version de Symfony. Pour plus de détails, lisez *Installation de Symfony*[4]

Pour télécharger les « vendor » plus rapidement, ajoutez l'option "--prefer-dist" à la fin de la commande « Composer ».

Cette commande peut prendre plusieurs minutes pour s'exécuter, car « Composer » télécharge la Distribution Standard ainsi que toutes les bibliothèques « vendor » dont elle a besoin. Lorsque la commande a terminé son exécution, vous devriez avoir un répertoire qui ressemble à quelque chose comme ça :

Listing 3-3
```
1  path/to/webroot/ <- votre répertoire racine
2      Symfony/ <- le nouveau répertoire
3          app/
4              cache/
5              config/
6              logs/
7          src/
8              ...
9          vendor/
10             ...
11         web/
12             app.php
13             ...
```

2. http://getcomposer.org/
3. http://getcomposer.org/download/
4. http://symfony.com/download

Option 2) Télécharger une archive

Vous pouvez aussi télécharger une archive contenant l'Édition Standard. Vous devrez alors faire deux choix :

- Télécharger l'archive au format `.tgz` ou `.zip`. Les deux sont équivalentes donc téléchargez celle avec laquelle vous vous sentez le plus à l'aise ;
- Téléchargez la distribution avec ou sans « vendors ». Si vous prévoyez d'utiliser d'autres bibliothèques tierces ou d'autres bundles et de les gérer via Composer, vous devriez probablement télécharger la distribution sans « vendors ».

Téléchargez l'une des archives quelque part dans le dossier racine de votre serveur web et extrayez-la. Depuis une interface de commande UNIX, cela peut être fait avec l'une des commandes suivantes (remplacez `###` par le nom du fichier) :

Listing 3-4
```
# pour l'archive .tgz
$ tar zxvf Symfony_Standard_Vendors_2.3.###.tgz

# pour l'archive .zip
$ unzip Symfony_Standard_Vendors_2.3.###.zip
```

Si vous avez téléchargé la distribution sans les « vendors », vous devez lire la section suivante.

Vous pouvez facilement surcharger la structure de répertoires par défaut. Lisez *Comment surcharger la structure de répertoires par défaut de Symfony* pour plus d'informations

Tous les fichiers publics et les contrôleurs frontaux qui prennent en charge les requêtes entrantes d'une application Symfony2 se trouvent dans le répertoire `Symfony/web/`. En conséquence, en supposant que vous avez extrait votre archive à la racine de votre serveur web ou de votre virtual host, l'URL de votre application commencera par `http://localhost/Symfony/web/`. Pour avoir des URLs plus courtes et plus agréables, vous devrez faire pointer la racine de votre serveur web vers le répertoire `Symfony/web/`. Bien que ce ne soit pas nécessaire pour développer, c'est recommandé pour une application en production car tous les fichiers système et liés à la configuration seront inaccessibles aux clients. Pour plus d'informations sur la manière de configurer vos serveurs web, lisez les documentations suivantes : *Apache*[5] | *Nginx*[6] .

Les exemples suivants supposent que vous n'avez pas touché la configuration de votre racine, donc toutes les urls commencent par `http://localhost/Symfony/web/`

Mettre à jour les Vendors

A ce stade, vous avez téléchargé un projet Symfony entièrement fonctionnel dans lequel vous allez commencer à développer votre propre application. Un projet Symfony dépend d'un certain nombre de bibliothèques externes. Celles-ci sont téléchargées dans le répertoire *vendor/* de votre projet via une bibliothèque appelée *Composer*[7].

Suivant la manière dont vous avez téléchargé Symfony, vous pourriez ou non avoir besoin de mettre à jour vos « vendors » dès maintenant. Mais, mettre à jour vos « vendors » est toujours sûr, et vous garantit d'avoir toutes les bibliothèques dont vous avez besoin.

5. `http://httpd.apache.org/docs/current/mod/core.html#documentroot`
6. `http://wiki.nginx.org/Symfony`
7. `http://getcomposer.org/`

Étape 1: Téléchargez *Composer*[8] (Le nouveau système de package PHP)

Listing 3-5
```
curl -s http://getcomposer.org/installer | php
```

Assurez-vous d'avoir téléchargé `composer.phar` dans le même répertoire que celui où se situe le fichier `composer.json` (par défaut à la racine de votre projet Symfony).

Étape 2: Installer les « vendors »

Listing 3-6
```
$ php composer.phar install
```

Cette commande télécharge toutes les bibliothèques nécessaires - incluant Symfony elle-même - dans le répertoire `vendor/`.

Si vous n'avez pas installé `curl`, vous pouvez juste télécharger le fichier `installer` manuellement à cette adresse *http://getcomposer.org/installer*[9]. Placez ce fichier dans votre projet puis lancez les commandes :

Listing 3-7
```
php installer
php composer.phar install
```

Lorsque vous exécutez `php composer.phar install` ou `php composer.phar update`, Composer va exécuter les commandes « install/update » pour vider le cache et installer les ressources (« assets » en anglais). Par défaut, les ressources seront copiées dans le répertoire `web`.

Au lieu de copier les ressources Symfony, vous pouvez créer des liens symboliques si votre système d'exploitation les supporte. Pour créer des liens symboliques, ajoutez une entrée dans le noeud `extra` de votre fichier composer.json avec la clé *symfony-assets-install`* et la valeur `symlink` :

Listing 3-8
```
"extra": {
    "symfony-app-dir": "app",
    "symfony-web-dir": "web",
    "symfony-assets-install": "symlink"
}
```

Si vous passez `relative` au lieu de `symlink` à la commande « symfony-assets-install », cette dernière génèrera des liens symboliques relatifs.

Configuration et installation

Maintenant, toutes les bibliothèques tierces nécessaires sont dans le répertoire `vendor/`. Vous avez également une application par défaut installée dans le répertoire `app/` et un exemple de code dans le répertoire `src/`.

Symfony2 est livré avec un testeur de configuration de votre serveur afin de vérifier que votre serveur web et PHP sont bien configuré pour utiliser Symfony. Utilisez l'URL suivante pour vérifier votre configuration :

Listing 3-9
```
http://localhost/config.php
```

S'il y a des problèmes, corrigez-les maintenant avant de poursuivre.

---

8. `http://getcomposer.org/`
9. `http://getcomposer.org/installer`

Définir les permissions

Un des problèmes les plus fréquents et que les répertoires `app/cache` et `app/logs` ne sont pas accessibles en écriture par le serveur web et par l'utilisateur de ligne de commande. Sur un système UNIX, si votre utilisateur de ligne de commande est différent de celui du serveur web, vous pouvez lancer les commandes suivantes une fois dans votre projet pour vous assurer que les permissions sont correctement définies.

Notez que tous les serveurs web n'utilisent pas l'utilisateur `www-data` comme dans les exemples ci-dessous. Veuillez vérifier quel utilisateur *votre* serveur web utilise et utilisez-le à la place de `www-data`.

Sur un système UNIX, vous pouvez le faire grâce à une des commandes suivantes :

Listing 3-10
```
$ ps aux | grep httpd
```

ou

Listing 3-11
```
$ ps aux | grep apache
```

1. Utiliser l'ACL sur un système qui supporte chmod +a

Beaucoup de systèmes autorisent l'usage de la commande `chmod +a`. Essayez d'abord la première méthode, et si vous avez une erreur, essayez la seconde. Assurez-vous de bien remplacer `www-data` par l'utilisateur de votre serveur web dans la première commande `chmod`:

Listing 3-12
```
$ rm -rf app/cache/*
$ rm -rf app/logs/*

$ sudo chmod +a "www-data allow delete,write,append,file_inherit,directory_inherit"
app/cache app/logs
$ sudo chmod +a "`whoami` allow delete,write,append,file_inherit,directory_inherit"
app/cache app/logs
```

2. Utiliser l'ACL sur un système qui ne supporte pas chmod +a

Certains systèmes ne supportent pas la commande``chmod +a``, mais supportent un autre utilitaire appelé `setfacl`. Vous devrez sans doute *activer le support ACL*[10] sur votre partition et installer setfacl avant de pouvoir l'utiliser (comme c'est le cas avec Ubuntu), de cette façon :

Listing 3-13
```
$ sudo setfacl -R -m u:www-data:rwX -m u:`whoami`:rwX app/cache app/logs
$ sudo setfacl -dR -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
```

3. Sans utiliser l'ACL

Si vous n'avez pas les droits de changer les accès aux répertoires, vous aurez besoin de changer le umask pour que les répertoires cache et log soit accessibles en écriture au groupe ou aux autres (cela dépend si l'utilisateur serveur web et l'utilisateur de ligne de commande sont dans le même groupe ou non). Pour faire ceci, ajoutez la ligne suivante au début des fichiers `app/console`, `web/app.php` et `web/app_dev.php`:

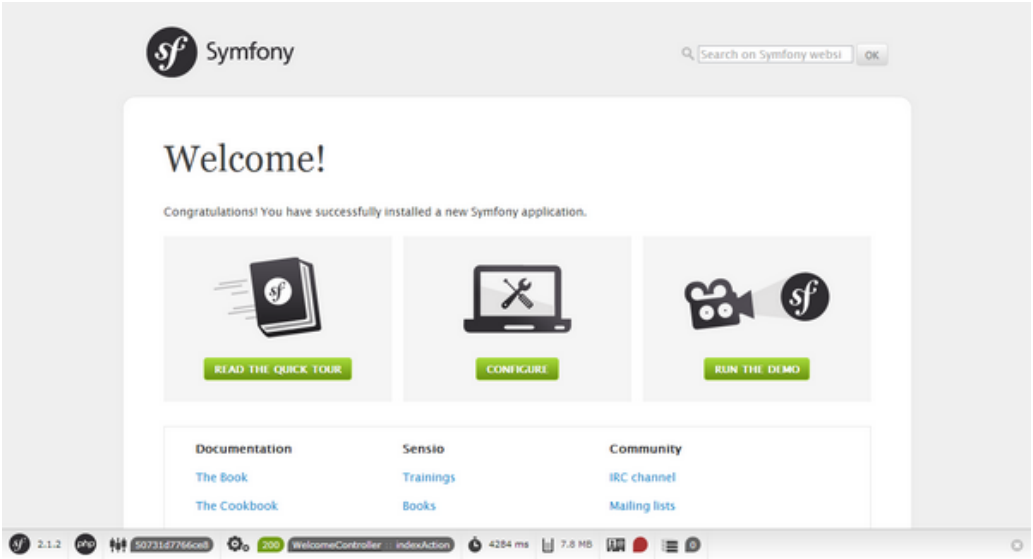
Listing 3-14
```
umask(0002); // Définit une permission 0775

// ou

umask(0000); // Définit une permission 0777
```

Notez qu'utiliser l'ALC est recommandé si vous y avez accès sur votre serveur car changer le umask n'est pas sûr.

Lorsque tout est bon, cliquez sur « Go to the Welcome page » pour afficher votre première « vraie » page Symfony2 :

Listing 3-15
```
1 http://localhost/app_dev.php/
```

Symfony2 devrait vous accueillir et vous féliciter pour tout le travail accompli jusqu'ici !



Commencer à développer

Maintenant que vous avez une application Symfony2 fonctionnelle, vous pouvez commencer à développer ! Votre distribution devrait contenir un exemple de code. Vérifiez le fichier `README.md` inclus avec la distribution (ouvrez-le en tant que fichier texte) pour savoir quel exemple de code est inclus avec votre distribution et savoir comment le supprimer par la suite.

Si vous découvrez Symfony, jetez un oeil au chapitre « *La création de pages avec Symfony2* », où vous apprendrez comment créer des pages, changer la configuration et faire tout ce que vous aurez besoin de faire dans votre nouvelle application.

Assurez-vous aussi de consulter le *Cookbook*, qui contient une grande variété d'articles expliquant comment solutionner des problèmes spécifiques avec Symfony.

Si vous voulez supprimer le code d'exemple de votre distribution, jetez un oeil à cet article du Cookbook : "`/cookbook/bundles/remove`"

Utiliser un Gestionnaire de Versions

Si vous utilisez un système de contrôle de version comme `Git` ou `Subversion`, vous pouvez le configurer et commencer à commiter votre projet normalement. La Symfony Standard edition *est* le point de départ de votre nouveau projet.

Pour des instructions spécifiques sur la meilleure façon de gérer votre projet avec git, lisez le chapitre *Comment créer et stocker un projet Symfony2 dans git*.

10. `https://help.ubuntu.com/community/FilePermissionsACLs`

Ignorer le répertoire `vendor/`

Si vous avez téléchargé l'archive *sans vendors*, vous pouvez ignorer tout le répertoire `vendor/` en toute sécurité et ne pas le commiter. Avec `Git`, cela se fait en créant le fichier `.gitignore` et en y ajoutant la ligne suivante:

Listing 3-16
```
1 /vendor/
```

Maintenant, le répertoire vendor ne sera pas commité sur votre système de gestion de code. C'est plutôt bien (en fait c'est génial !) car lorsque quelqu'un clone ou récupère le projet, il lui suffit de lancer la commande `php bin/vendors install` pour récupérer toutes les bibliothèques nécessaires.

Chapter 4
La création de pages avec Symfony2

Créer des pages avec Symfony2 se fait en simplement deux étapes :

- *Créez une route* : Une route définit l'URL (ex: `/apropos`) pour votre page et spécifie un contrôleur (une fonction PHP) que Symfony2 devrait exécuter quand l'URL d'une requête HTTP correspond à une route que vous avez définie.
- *Créez un contrôleur* : Un contrôleur est une fonction PHP qui traitera la requête HTTP et la transformera en un objet `Response` Symfony2 qui sera retourné à l'utilisateur.

Cette approche très simple est excellente, car elle correspond à la façon dont fonctionne le Web. Chaque interaction sur le Web est initiée par une requête HTTP. Le but de votre application est simplement d'interpréter cette requête et de lui retourner une Response HTTP appropriée.

Symfony2 suit cette philosophie et vous fournit des outils et conventions pour garder votre application organisée tout en pouvant devenir plus complexe et fréquentée.

Cela vous parait suffisamment simple ? Alors, allons-y !

La page « Hello Symfony ! »

Commençons avec une application traditionnelle « Hello World ! ». Quand nous aurons terminé, l'utilisateur sera capable de recevoir un message de salutation personnalisé (ex « Hello Symfony ») en se rendant à l'URL suivante :

Listing 4-1
```
1 http://localhost/app_dev.php/hello/Symfony
```

En fait, vous serez capable de pouvoir remplacer `Symfony` par n'importe quel autre nom qui doit être salué. Afin de créer cette page, suivez ce simple processus en deux étapes.

Ce tutoriel suppose que vous ayez déjà téléchargé et configuré Symfony2 sur votre serveur web. L'URL ci-dessus suppose que `localhost` pointe vers le répertoire `web` de votre nouveau projet Symfony2. Pour des instructions détaillées à ce sujet, lisez la documentation du serveur web que vous utilisez. Voici les documentations de serveur que vous utilisez probablement :

- Pour Apache HTTP Server, allez voir la *documentation Apache's DirectoryIndex*[1].
- Pour Nginx, allez voir *documentation Nginx HttpCoreModule*[2].

Avant de commencer : Créez un Bundle

Avant de commencer, vous devez créer un *bundle*. Dans Symfony2, un *bundle* est comme un plugin, excepté le fait que tout le code de votre application siégera dans un bundle.

Un bundle n'est rien d'autre qu'un répertoire qui contient tout ce qui est relatif à une fonctionnalité spécifique, ce qui inclut les classes PHP, la configuration et même les feuilles de style et le javascript (voir *Le Système de Bundles*).

Afin de créer un bundle nommé `AcmeHelloBundle` (un bundle fictif que vous créerez dans ce chapitre), lancez la commande suivante et suivez les instructions affichées à l'écran (choisissez les options par défaut) :

Listing 4-2

```
$ php app/console generate:bundle --namespace=Acme/HelloBundle --format=yml
```

Un répertoire `src/Acme/HelloBundle` est créé pour le bundle. Une ligne est aussi automatiquement ajoutée au fichier `app/AppKernel.php` afin que le bundle soit correctement enregistré:

Listing 4-3

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Acme\HelloBundle\AcmeHelloBundle(),
    );
    // ...

    return $bundles;
}
```

Maintenant que votre bundle est mis en place, vous pouvez commencer à construire votre application à l'intérieur du bundle.

Etape 1 : Créez la Route

Dans une application Symfony2, le fichier de configuration des routes se trouve par défaut dans `app/config/routing.yml`. Comme toute configuration dans Symfony2, vous pouvez également choisir d'utiliser des fichiers XML ou PHP afin de configurer vos routes.

Si vous regardez le fichier de routage principal, vous verrez que Symfony a déjà ajouté une entrée lorsque vous avez généré le `AcmeHelloBundle` :

Listing 4-4

1. http://httpd.apache.org/docs/2.0/mod/mod_dir.html
2. http://wiki.nginx.org/HttpCoreModule#location

```yaml
# app/config/routing.yml
 acme_hello:
     resource: "@AcmeHelloBundle/Resources/config/routing.yml"
     prefix:   /
```

Ce code est très basique : il dit à Symfony de charger la configuration de routage depuis le fichier `Resources/config/routing.yml` qui se trouve dans le `AcmeHelloBundle`. Cela signifie que vous pouvez placer votre configuration de routage directement dans le fichier `app/config/routing.yml` ou organiser vos routes dans votre application et les importer depuis ce fichier.

Maintenant que le fichier `routing.yml` du bundle est importé, ajoutez la nouvelle route qui définit l'URL de la page que vous êtes sur le point de créer :

Listing 4-5
```yaml
# src/Acme/HelloBundle/Resources/config/routing.yml
hello:
    pattern:  /hello/{name}
    defaults: { _controller: AcmeHelloBundle:Hello:index }
```

Le routage est constitué de deux parties principales : le `pattern`, qui est l'URL correspondante à cette route, et un tableau `par  défaut`, qui spécifie le contrôleur qui devra être exécuté. La syntaxe pour le paramètre dans le pattern (`{name}`) est un joker. Cela signifie que `hello/Ryan`, `hello/Bernard` ou n'importe quelle URL similaire correspondra à cette route. Le paramètre `{name}` sera également passé à notre contrôleur afin que nous puissions utiliser la valeur afin de saluer l'utilisateur.

Le système de routage dispose de beaucoup d'autres fonctionnalités qui vous permettront de créer des structures d'URL puissantes et flexibles dans votre application. Pour plus de détails, lisez le chapitre dédié aux routes *Routing*.

Etape 2 : Créez le Contrôleur

Quand une URL comme `/hello/Ryan` est traitée par l'application, la route `hello` est reconnue et le contrôleur `AcmeHelloBundle:Hello/index` est exécuté par le framework. L'étape suivante est de créer ce contrôleur.

Le contrôleur - `AcmeHelloBundle:Hello:index` est le nom *logique* du contrôleur, et il est associé à la méthode `indexAction` d'une classe PHP appelée `Acme\HelloBundle\Controller\HelloController`. Commencez par créer ce fichier dans votre `AcmeHelloBundle` :

Listing 4-6
```php
// src/Acme/HelloBundle/Controller/HelloController.php
namespace Acme\HelloBundle\Controller;

class HelloController
{
}
```

En réalité, un contrôleur n'est rien d'autre qu'une méthode PHP que vous créez et que Symfony exécute. C'est à cet endroit que le code propre à l'application utilisera les informations de la requête afin de construire et préparer la ressource demandée par la requête. Excepté dans certaines situations avancées, le résultat final d'un contrôleur sera toujours le même : un objet `Response` Symfony2.

Créez la méthode `indexAction` que Symfony exécutera lorsque la route `hello` sera identifiée :

Listing 4-7
```php
// src/Acme/HelloBundle/Controller/HelloController.php
namespace Acme\HelloBundle\Controller;
```

```
3
4  use Symfony\Component\HttpFoundation\Response;
5
6  class HelloController
7  {
8      public function indexAction($name)
9      {
10         return new Response('<html><body>Hello '.$name.'!</body></html>');
11     }
12 }
```

Le contrôleur est simple : il crée un nouvel objet `Response` qui a pour premier argument le contenu qui doit être utilisé dans la réponse (une petite page HTML dans notre cas).

Félicitations ! Après avoir n'avoir créé qu'une route et un contrôleur, vous avez une page pleinement fonctionnelle ! Si vous avez tout effectué correctement, votre application devrait vous saluer:

Listing 4-8
```
1 http://localhost/app_dev.php/hello/Ryan
```

Vous pouvez aussi voir votre application en *environnement* de « prod » en allant à l'adresse :

Listing 4-9
```
1 http://localhost/app.php/hello/Ryan
```

Si vous avez une erreur, c'est certainement parce que vous avez besoin de vider votre cache grâce à la commande :

Listing 4-10
```
1 $ php app/console cache:clear --env=prod --no-debug
```

Une troisième étape optionnelle dans ce processus est de créer un template.

Les contrôleurs sont le point central de votre code et un élément clé de la création de pages. Pour plus d'informations lisez le chapitre *Chapitre Contrôleurs*.

Étape 3 facultative : Créez le Template

Les templates vous permettent de déplacer toute la présentation (ex: code HTML) dans un fichier séparé et de réutiliser différentes portions d'un layout. A la place d'écrire le code HTML dans le contrôleur, retournez plutôt un template :

Listing 4-11
```
1  // src/Acme/HelloBundle/Controller/HelloController.php
2
3  namespace Acme\HelloBundle\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6
7  class HelloController extends Controller
8  {
9      public function indexAction($name)
10     {
11         return $this->render('AcmeHelloBundle:Hello:index.html.twig', array('name' =>
12 $name));
```

```
13
14          // render a PHP template instead
15          // return $this->render('AcmeHelloBundle:Hello:index.html.php', array('name' =>
16  $name));
        }
    }
```

Afin d'utiliser la méthode *render()*[3], votre contrôleur doit étendre la classe *SymfonyBundleFrameworkBundleControllerController`* (API docs: *Controller*[4]), qui ajoute des raccourcis pour des tâches fréquemment utilisées dans les contrôleurs. Dans l'exemple ci-dessus, c'est ce qui est fait en ajoutant la ligne `use` et en étendant la classe `Controller` à la ligne 6.

La méthode `render()` crée un objet `Response` qui contient le contenu d'un template rendu. Comme tout autre contrôleur, vous retournerez cet objet `Response`.

Notez qu'il y a deux différents exemples afin de rendre un template. Par défaut, Symfony2 supporte deux langages différents de templates : les templates classiques PHP et les simples, mais puissants templates `Twig`. Ne paniquez pas, vous êtes libres de choisir celui que vous désirez voire les deux.

Le contrôleur rend le template `AcmeHelloBundle:Hello:index.html.twig`, qui suit la convention de nommage :

NomBundle:**NomContrôleur**:**NomTemplate**

C'est le nom *logique* du template, qui est associé à une location physique selon la convention suivante.

/chemin/vers/NomBundle/Resources/views/**NomContrôleur**/**NomTemplate**

Dans ce cas, `AcmeHelloBundle` est le nom de bundle, `Hello` est le nom du contrôleur et enfin `index.html.twig` est le template :

Listing 4-12
```
1  {# src/Acme/HelloBundle/Resources/views/Hello/index.html.twig #}
2  {% extends '::base.html.twig' %}
3
4  {% block body %}
5      Hello {{ name }}!
6  {% endblock %}
```

Analysons maintenant le template Twig ligne par ligne :

- *Ligne 2* : Le symbole `extends` définit un template parent. Le template définit explicitement un fichier layout dans lequel il sera inséré.
- *Ligne 4* : Le symbole `block` indique que tout ce qui est à l'intérieur doit être placé à l'intérieur d'un bloc appelé `body`. Comme vous le voyez, c'est en définitive la responsabilité du template parent (`base.html.twig`) de rendre le bloc `body`.

Le nom de fichier du template parent, `::base.html.twig`, est dispensé des portions **NomBundle** et **NomContrôleur** (remarquez les deux points (`::`) au début). Ceci signifie que le template se situe en dehors du bundle et dans le répertoire `app`.

Listing 4-13
```
1  {# app/Resources/views/base.html.twig #}
2  <!DOCTYPE html>
```

3. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#render()
4. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle/Controller/Controller.html

```twig
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
        <link rel="shortcut icon" href="{{ asset('favicon.ico') }}" />
    </head>
    <body>
        {% block body %}{% endblock %}
{% block javascripts %}{% endblock %}
    </body>
</html>
```

Le fichier du template de base définit le layout HTML en rend le bloc `body` que vous avez défini dans le template `index.html.twig`. Il rend également un bloc `title`, que vous pouvez choisir de définir dans le template `index.html.twig`. Si vous ne définissez pas le bloc `title` dans le template enfant, il aura pour valeur par défaut `Welcome!`.

Les templates sont une façon puissante de rendre et d'organiser le contenu de votre page. Les templates peuvent tout rendre, des layouts HTML aux codes CSS, ou n'importe quoi d'autre que le contrôleur peut avoir besoin de retourner à l'utilisateur.

Dans le cycle de vie d'une requête, le template est un outil facultatif. Souvenez vous que le but de chaque contrôleur est de renvoyer un objet `Response`. Le moteur de templates est un outil puissant, bien qu'optionnel, pour créer le contenu de cet objet `Response`.

La structure des répertoires

Après seulement quelques courtes sections, vous comprenez déjà la philosophie derrière la création et le rendu de pages dans Symfony2. Vous avez déjà commencé à voir comment les projets Symfony2 sont structurés et organisés. A la fin de cette section, vous saurez où trouver et où mettre les différents types de fichiers et pourquoi.

Bien qu'entièrement flexible, chaque *application* a par défaut la même structure de répertoires basique et recommandée :

- `app/`: Ce répertoire contient la configuration de l'application;
- `src/`: Tout le code PHP du projet est stocké ici;
- `vendor/`: Par convention, toutes les bibliothèques tierces (additionnelles) sont placées ici;
- `web/`: Le répertoire racine web qui contient tous les fichiers publiquement accessibles;

Le répertoire Web

Le répertoire web contient tous les fichiers publics et statiques incluant les images, les feuilles de style et les javascripts. Il contient également le « *contrôleur frontal* » (« front controller » en anglais) :

Listing 4-14
```php
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

Le contrôleur frontal (`app.php` dans cet exemple) est le fichier exécuté lorsque l'on appelle une application Symfony2. Son rôle est d'utiliser une classe Kernel, `AppKernel`, pour initialiser l'application (bootstrap).

Avoir un contrôleur frontal signifie des URL différentes et plus flexibles que dans une application en pur php. Lorsqu'on utilise un contrôleur frontal, les URLs sont formatées comme suit :

Listing 4-15
```
1 http://localhost/app.php/hello/Ryan
```

Le contrôleur frontal, `app.php`, est exécuté et l'URL « interne: » `/hello/Ryan` est traitée par l'application en se basant sur la configuration du routage. En utilisant les règles du module Apache `mod_rewrite`, vous pouvez forcer le script `app.php` à être exécuté sans avoir besoin de le mentionner dans l'URL:

Listing 4-16
```
1 .. code-block:: text
2
3 http://localhost/hello/Ryan
```

Les contrôleurs frontaux sont essentiels pour traiter chaque requête. Cependant, vous aurez rarement besoin de les modifier ou même d'y penser. Ils seront abordés dans la section Environnements .

Le répertoire de l'application (app)

Comme nous l'avons vu dans le contrôleur frontal, la classe `AppKernel` est le point d'entrée principal de l'application et est chargée de toute la configuration. De ce fait, elle est stockée dans le répertoire `app/`.

Cette classe doit implémenter deux méthodes qui définissent tout ce dont Symfony a besoin de savoir à propos de votre application. Vous n'avez pas à vous soucier de ces méthodes en commençant - Symfony les complète pour vous avec des valeurs par défaut.

- `registerBundles()`: renvoit un tableau de tous les bundles dont l'application a besoin pour fonctionner (voir le *Le Système de Bundles*) ;
- **`registerContainerConfiguration()`: Charge le fichier de configuration ressource**

  principal de l'application (voir la section Configuration de l'Application).

Dans le développement au quotidien, vous utiliserez principalement le répertoire `app/` pour modifier les fichiers de configuration et de routage dans le répertoire `app/config` (voir Configuration de l'Application). `app/` contient également le répertoire de cache de l'application (`app/cache`), le répertoire des logs (`app/logs`) et un répertoire pour les ressources communes à l'application entière (`app/Resources`). Nous en apprendrons plus sur ces répertoires dans de prochains chapitres.

Autoloading

Lorsque Symfony se charge, un fichier spécial - `app/autoload.php` - est inclus. Ce fichier s'occupe de configurer l'autoloader qui chargera automatiquement tous vos fichiers depuis le répertoire `src/` et toutes les bibliothèques tierces depuis le repertoire `vendor/`.

Grâce à l'autoloader, vous n'avez jamais à vous soucier d'utiliser les instructions `include` ou `require`. Symfony2 se base sur l'espace de nom (namespace) d'une classe pour déterminer son emplacement et l'inclure automatiquement le fichier à votre place à l'instant où vous en avez besoin.

L'autoloader est déjà configuré pour regarder dans le dossier `src/` pour chacune de vos classes PHP. Pour que le chargement automatique fonctionne, le nom de la classe et le chemin du fichier doivent avoir une structure similaire :

Listing 4-17

```
Nom de classe:
    Acme\HelloBundle\Controller\HelloController
Chemin:
    src/Acme/HelloBundle/Controller/HelloController.php
```

Typiquement, le seul moment où vous devrez vous soucier du fichier `app/autoload.php` est quand vous inclurez des bibliothèques tierces dans le répertoire `vendor/`. Pour plus d'informations sur le chargement automatique, voir :doc: *Comment charger automatiquement des classes</components/class_loader>*.

Le répertoire des sources (`src/`)

Pour faire simple, le répertoire `src/` contient tout le code (code PHP, templates, fichiers de configuration, feuilles de style, etc) qui fait tourner *votre* application. En fait en développant, le plus gros du travail sera fait à l'intérieur d'un ou plusieurs bundles que vous créerez dans ce répertoire.

Mais qu'est-ce qu'un *bundle*?

Le Système de Bundles

Un bundle est similaire aux plugins que l'on peut trouver dans d'autres logiciels, mais en mieux. La différence clé est que *tout* est un bundle dans Symfony2, ce qui inclut le coeur du framework et le code de votre application. Les bundles sont aux premières loges dans Symfony2. Ils vous offrent la flexibilité d'utiliser des fonctionnalités préconstruites packagées dans des *bundles tiers*[5] ou de distribuer vos propres bundles. Cela rend facile de sélectionner quelles fonctionnalités activer dans votre application et de les optimiser comme vous voulez.

Alors que vous apprendrez les bases ici, une entrée du cookbook est entièrement dédiée à l'organisation et aux bonnes pratiques : *bundles*.

Un bundle est simplement un ensemble structuré de fichiers au sein d'un répertoire et qui implémentent une fonctionnalité unique. Vous pourrez ainsi créer un `BlogBundle`, un `ForumBundle` ou un bundle pour la gestion des utilisateurs (beaucoup de ces bundles existent déjà et sont open source). Chaque répertoire contient tout ce qui est lié à cette fonctionnalité incluant les fichiers PHP, les templates, les feuilles de style, le javascript, les tests et tout le reste. Chaque aspect d'une fonctionnalité se trouve dans le bundle, et chaque fonctionnalité aussi.

5. `http://knpbundles.com`

Une application est faite de bundles qui sont définis dans la méthode `registerBundles()` de la classe `AppKernel` :

Listing 4-18
```php
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
    );

    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
        $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }

    return $bundles;
}
```

Avec la méthode `registerBundles()`, vous avez le contrôle totale des bundles qui sont utilisés dans votre application (incluant les bundles du coeur de Symfony).

Un bundle peut se trouver *n'importe où* pour peu qu'il puisse être chargé (via l'autoloader configuré à `app/autoload.php`).

Créer un Bundle

La Symfony Standard Edition est fournie avec une tâche qui crée un bundle totalement fonctionnel pour vous. Bien sûr, vous pouvez tout aussi facilement créer un bundle à la main.

Pour vous montrer à quel point le système de bundle est simple, créons un nouveau bundle appelé `AcmeTestBundle` et activons-le.

La partie `Acme` est juste un nom idiot qui peut être remplacé par un autre nom qui vous représente ou votre entreprise (ex `ABCTestBundle` pour une entreprise nommée `ABC`).

Commencez par créer un répertoire `src/Acme/TestBundle/` et ajoutez y un nouveau fichier appelé `AcmeTestBundle.php` :

Listing 4-19
```php
// src/Acme/TestBundle/AcmeTestBundle.php
namespace Acme\TestBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

```

```
6  class AcmeTestBundle extends Bundle
7  {
8  }
```

Le nom `AcmeTestBundle` suit les *conventions de nommage des bundles*. Vous pourriez aussi choisir de raccourcir le nom du bundle pour `TestBundle` en nommant sa classe `TestBundle` (et en appelant le fichier `TestBundle.php`).

Cette classe vide est la seule pièce dont vous avez besoin afin de créer un nouveau bundle. Bien que souvent vide, cette classe est très puissante et peut être utilisée pour personnaliser le comportement du bundle.

Maintenant que vous avez créé le bundle, activez-le via la classe `AppKernel` :

Listing 4-20
```
1  // app/AppKernel.php
2  public function registerBundles()
3  {
4      $bundles = array(
5          // ...
6
7          // register your bundles
8          new Acme\TestBundle\AcmeTestBundle(),
9      );
10     // ...
11
12     return $bundles;
13 }
```

Et même s'il ne fait encore rien de spécial, `AcmeTestBundle` est prêt à être utilisé.

Et bien que ce soit très facile, Symfony fournit également une commande qui génère un squelette de bundle de base :

Listing 4-21
```
1  $ php app/console generate:bundle --namespace=Acme/TestBundle
```

Le squelette du bundle contient un contrôleur de base, un template et une configuration de routage qui peuvent être personnalisés. Vous en apprendrez plus sur les commandes Symfony2 plus tard.

Peu importe que vous créiez un bundle ou que vous utilisiez un bundle tiers, assurez-vous toujours qu'il soit activé dans `registerBundles()`. Si vous utilisez la commande `generate:bundle`, c'est fait automatiquement pour vous.

Structure des répertoires des bundles

La structure des répertoires d'un bundle est simple et flexible. Par défaut le système de bundle suit un ensemble de conventions qui aident à garder le code homogène entre tous les bundles Symfony2. Jetez un oeil au `AcmeHelloBundle`, car il contient certains des éléments les plus communs d'un bundle :

- `Controller/` contient les contrôleurs du bundle (ex `HelloController.php`);
- `DependencyInjection/` contient certaines classes d'extension d'injection de dépendances, qui peuvent importer des configurations de services, enregistrer des passes de compilation ou plus encore (ce répertoire n'est pas obligatoire);

- `Resources/config/` contient la configuration, notamment la configuration de routage (ex `routing.yml`);
- `Resources/views/` contient les templates organisés par nom de contrôleur (ex `Hello/index.html.twig`);
- `Resources/public/` contient les ressources web (images, feuilles de style, etc) et sont copiées ou liées par un lien symbolique dans le répertoire de projet `web/` grâce à la commande `assets:install`;
- `Tests/` contient tous les tests du bundle.

Un bundle peut être très petit ou très grand selon la fonctionnalité qu'il implémente. Il contient seulement les fichiers dont vous avez besoin et rien d'autre.

En parcourant le Book, vous apprendrez comment persister des objets en base de données, créer et valider des formulaires, créer des traductions pour votre application, écrire des tests et bien plus encore. Chacun de ces aspects a sa propre place et son propre rôle au sein d'un bundle.

Configuration de l'Application

Une application consiste en un ensemble de bundles qui représente toutes les fonctionnalités et capacités de votre application. Chaque bundle peut être personnalisé via les fichiers de configuration écrits en YAML, XML ou PHP. Par défaut, la configuration principale se situe dans le répertoire `app/config/` et se trouve dans un fichier appelé `config.yml`, `config.xml` ou `config.php` selon le format que vous préférez :

Listing 4-22
```yaml
# app/config/config.yml
imports:
    - { resource: parameters.ini }
    - { resource: security.yml }

framework:
    secret:          "%secret%"
    router:          { resource: "%kernel.root_dir%/config/routing.yml" }
    #...

# Twig Configuration
twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"

# ...
```

Vous apprendrez exactement comment charger chaque fichier/format dans la prochaine section Environnements.

Chaque entrée de niveau zéro comme `framework` ou `twig` définit la configuration d'un bundle particulier. Par exemple, la clé `framework` définit la configuration du bundle du noyau de Symfony `FrameworkBundle` et inclut la configuration pour le routage, les templates et d'autres fonctionnalités du noyau.

Pour le moment, ne vous inquiétez pas des options de configuration spécifiques à chaque section. Le fichier de configuration a des valeurs par défaut optimisées. Au fur et à mesure que vous lirez et explorerez chaque recoin de Symfony2, vous en apprendrez plus sur les options de configuration spécifiques à chaque fonctionnalité.

Formats de Configuration

A travers les chapitres, tous les exemples de configuration seront montrés dans les trois formats (YAML, XML and PHP). Chacun a ses avantages et ses inconvénients. Le choix vous appartient :

- *YAML*: Simple, propre et lisible;
- *XML*: Plus puissant que YAML parfois et support de l'autocomplétion sur les IDE;
- *PHP*: Très puissant, mais moins lisible que les formats de configuration standards.

Dump de configuration par défaut

. versionadded:: 2.1
La commande `config:dump-reference` a été ajoutée dans la version 2.1 de Symfony

Vous pouvez dumper la configuration par défaut d'un bundle en yaml vers la console en utilisant la commande `config:dump-reference`. Voici un exemple de dump de la configuration du FrameworkBundle :

Listing 4-23
```
1 app/console config:dump-reference FrameworkBundle
```

L'alias de l'extension (clé de configuration) peut aussi être utilisé :

Listing 4-24
```
1 app/console config:dump-reference framework
```

Lisez l'article du Cookbook : *Comment exposer une configuration sémantique pour un Bundle* pour avoir des informations sur l'ajout de configuration dans votre bundle.

Environnements

Une application peut tourner sous différents environnements. Les différents environnements partagent le même code PHP (excepté celui du contrôleur frontal), mais utilisent une configuration différente. Par exemple, l'environnement de `dev` enregistrera les erreurs et les warnings dans les logs, tandis que l'environnement de `prod` enregistrera seulement les erreurs. Certains fichiers sont reconstruits à chaque requête en environnement de `dev` (pour rendre le développement plus pratique), mais sont mis en cache en environnement de `prod`. Tous les environnements peuvent tourner ensemble sur la même machine et exécutent la même application.

Un projet Symfony2 commence en général avec 3 environnements (`dev`, `test` et `prod`), la création d'un nouvel environnement étant très facile. Vous pouvez voir l'application sous différents environnements en changeant simplement le contrôleur frontal dans votre navigateur. Pour voir l'application en environnement de `dev`, accédez à l'application via le contrôleur frontal de développement :

Listing 4-25
```
1 http://localhost/app_dev.php/hello/Ryan
```

Si vous désirez voir comment votre application se comporterait en environnement de production, appelez le contrôleur frontal de `prod` :

Listing 4-26
```
1 http://localhost/app.php/hello/Ryan
```

Puisque l'environnement de `prod` est optimisé pour la vitesse; la configuration, les routes et les templates Twig sont compilés en classes PHP et cachés. Quand vous voudrez voir des changements en environnement de `prod`, vous aurez besoin de nettoyer ces fichiers cachés afin de permettre leur régénération :

Listing 4-27
```
$ php app/console cache:clear --env=prod --no-debug
```

Si vous ouvrez le fichier `web/app.php`, vous trouverez ce qui est explicitement configuré en environnement de `prod` :

Listing 4-28
```
$kernel = new AppKernel('prod', false);
```

Vous pouvez créer un nouveau contrôleur frontal pour un nouvel environnement en copiant ce fichier et en changeant `prod` par une autre valeur.

L'environnement de `test` est utilisé pour lancer des tests automatiques et n'est pas accessible directement dans un navigateur. Lisez le chapitre *tests* pour plus de détails.

Configuration d'Environnement

La classe `AppKernel` est responsable du chargement du fichier de configuration que vous avez choisi :

Listing 4-29
```
// app/AppKernel.php
public function registerContainerConfiguration(LoaderInterface $loader)
{
    $loader->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
}
```

Vous savez déjà que l'extension `.yml` peut être changée en `.xml` ou `.php` si vous préférez utiliser le format XML ou PHP pour écrire votre configuration. Notez également que chaque environnement charge sa propre configuration. Considérez le fichier de configuration pour l'environnement de `dev`.

Listing 4-30
```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

framework:
    router:   { resource: "%kernel.root_dir%/config/routing_dev.yml" }
    profiler: { only_exceptions: false }

# ...
```

La clé `imports` est similaire à l'instruction PHP `include` et garantit que le fichier de configuration principal (`config.yml`) est chargé en premier. Le reste du fichier modifie la configuration par défaut pour une augmentation des logs et d'autres paramètres relatifs à un environnement de développement.

Les environnements de `prod` et de `test` suivent le même modèle : chaque environnement importe le fichier de configuration de base et modifie ses valeurs pour s'adapter aux besoins spécifiques à l'environnement. C'est juste une convention, mais elle vous permet de réutiliser la plupart de la configuration et d'en modifier une partie en fonction de l'environnement.

Résumé

Félicitations ! Vous avez maintenant eu un aperçu de chaque aspect fondamental de Symfony2 et avez découvert sa facilité et sa flexibilité. Et même s'il y a encore *beaucoup* de fonctionnalité à découvrir, gardez les principes de base suivants en tête :

- créer une page est un processus en trois étapes impliquant une **route**, un **contrôleur** et (optionnellement) un **template**.
- chaque projet contient juste quelques répertoires principaux : `web/` (ressources web et contrôleurs frontaux), `app/` (configuration), `src/` (vos bundles), et `vendor/` (bibliothèques tierces) (il y a aussi un répertoire `bin/` qui est utilisé pour la mise à jour des bibliothèques vendors);
- chaque fonctionnalité de Symfony2 (incluant le noyau du framework) est organisée dans un *bundle*, qui est un ensemble structuré de fichiers pour cette fonctionnalité;
- la **configuration** de chaque bundle se trouve dans le répertoire `Ressources/config` du bundle et peut être écrite en YAML, XML ou PHP;
- La **configuration globale de l'application** se trouve dans le répertoire `app/config`;
- chaque **environnement** est accessible via des contrôleurs frontaux différents (ex `app.php` et `app_dev.php`) et charge un fichier de configuration différent.

A partir de maintenant, chaque chapitre vous introduira de plus en plus d'outils puissants et de concepts avancés. Plus vous connaîtrez Symfony2, plus vous apprécierez la flexibilité de son architecture et le pouvoir qu'il vous donne pour développer rapidement des applications.

Chapter 5
Le Contrôleur

Un contrôleur est une fonction PHP que vous créez pour prendre les informations provenant de la requête HTTP et qui construit puis retourne une réponse HTTP (sous forme d'un objet Symfony2 `Response`). La réponse peut être une page HTML, un document XML, un tableau JSON sérialisé, une image, une redirection, une erreur 404 ou tout ce que vous pouvez imaginer. Le contrôleur contient n'importe quelle logique arbitraire dont *votre application* a besoin pour retourner le contenu d'une page.

Pour en illustrer la simplicité, jetons un oeil à un contrôleur Symfony2 en action. Le contrôleur suivant affiche une page qui écrit simplement `Hello world!`:

Listing 5-1

```
use Symfony\Component\HttpFoundation\Response;

public function helloAction()
{
    return new Response('Hello world!');
}
```

Le but d'un contrôleur est toujours le même : créer et retourner un objet `Response`. Durant ce processus, il peut lire des informations dans la requête, charger une ressource depuis la base de données, envoyer un email, ou définir une variable dans la session de l'utilisateur. Mais dans tous les cas, le contrôleur va finalement retourner l'objet `Response` qui sera délivré au client.

Il n'y a pas de magie et aucune autre exigence à prendre en compte ! Voici quelques exemples classiques :

- *Le contrôleur A* prépare un objet `Response` représentant le contenu de la page d'accueil.
- *Le contrôleur B* lit le paramètre `slug` contenu dans la requête pour charger une entrée du blog depuis la base de données et crée un objet `Response` affichant ce blog. Si le `slug` ne peut pas être trouvé dans la base de données, il crée et retourne un objet `Response` avec un code de statut 404.
- *Le contrôleur C* gère la soumission d'un formulaire de contact. Il lit les informations de la requête, enregistre les informations du contact dans la base de données et envoie ces dernières par email au webmaster. Enfin, il crée un objet `Response` qui redirige le navigateur du client vers la page « merci » du formulaire de contact.

Cycle de vie Requête, Contrôleur, Réponse

Chaque requête gérée par un projet Symfony2 suit le même cycle de vie. Le framework s'occupe des tâches répétitives et exécute finalement un contrôleur qui contient votre code applicatif personnalisé :

1. Chaque requête est gérée par un unique fichier contrôleur frontal (par exemple: `app.php` ou `app_dev.php`) qui initialise l'application;
2. Le `Router` lit l'information depuis la requête (par exemple: l'URI), trouve une route qui correspond à cette information, et lit le paramètre `_controller` depuis la route;
3. Le contrôleur correspondant à la route est exécuté et le code interne au contrôleur crée et retourne un objet `Response`;
4. Les en-têtes HTTP et le contenu de l'objet `Response` sont renvoyés au client.

Créer une page est aussi facile que de créer un contrôleur (#3) et d'implémenter une route qui y fasse correspondre une URL (#2).

Bien que son nom est très similaire, un « contrôleur frontal » est différent des « contrôleurs » abordés dans ce chapitre. Un contrôleur frontal est un petit fichier PHP qui se situe dans votre répertoire web et à travers lequel toutes les requêtes sont dirigées. Une application typique va avoir un contrôleur frontal de production (par exemple: `app.php`) et un contrôleur frontal de développement (par exemple: `app_dev.php`). Vous n'aurez vraisemblablement jamais besoin d'éditer, de regarder ou de vous occuper des contrôleurs frontaux dans votre application.

Un contrôleur simple

Bien qu'un contrôleur puisse être n'importe quelle « chose PHP » appelable (une fonction, une méthode d'un objet, ou une `Closure`), dans Symfony2, un contrôleur est généralement une unique méthode à l'intérieur d'un objet contrôleur. Les contrôleurs sont aussi appelés *actions*.

Listing 5-2

```
1  // src/Acme/HelloBundle/Controller/HelloController.php
2  namespace Acme\HelloBundle\Controller;
3
4  use Symfony\Component\HttpFoundation\Response;
5
6  class HelloController
7  {
8      public function indexAction($name)
9      {
10         return new Response('<html><body>Hello '.$name.'!</body></html>');
11     }
12 }
```

Notez que le *contrôleur* est la méthode `indexAction`, qui réside dans une *classe contrôleur* (`HelloController`). Ne soyez pas gêné par ce nom : une *classe contrôleur* est simplement une manière pratique de grouper plusieurs contrôleurs/actions ensemble. Typiquement, la classe contrôleur va héberger plusieurs contrôleurs/actions (par exemple : `updateAction`, `deleteAction`, etc).

Ce contrôleur est relativement simple :

- *ligne 4*: Symfony2 tire avantage de la fonctionnalité des espaces de noms (« namespaces ») de PHP 5.3 afin de donner un espace de noms à la classe entière du contrôleur. Le mot-clé `use` importe la classe `Response`, que notre contrôleur doit retourner.

- *ligne 6*: Le nom de la classe est la concaténation d'un nom pour la classe du contrôleur (par exemple: `Hello`) et du mot `Controller`. Ceci est une convention qui fournit une uniformité aux contrôleurs et qui leur permet d'être référencés seulement par la première partie du nom (par exemple: `Hello`) dans la configuration de routage (« routing »).
- *ligne 8*: Chaque action d'une classe contrôleur se termine par `Action` et est référencée dans la configuration de routage par le nom de l'action (ex `index`). Dans la prochaine section, vous allez créer une route qui fait correspondre une URI à son action. Vous allez apprendre comment les paramètres de la route (par exemple `{name}`) deviennent les arguments de la méthode action (`$name`).
- *ligne 10*: Le contrôleur crée et retourne un objet `Response`.

Faire correspondre une URL à un Contrôleur

Le nouveau contrôleur retourne une simple page HTML. Pour voir cette page dans votre navigateur, vous avez besoin de créer une route qui va faire correspondre un pattern d'URL spécifique à ce contrôleur :

Listing 5-3
```yaml
# app/config/routing.yml
hello:
    path:      /hello/{name}
    defaults:  { _controller: AcmeHelloBundle:Hello:index }
```

Aller à l'URL `/hello/ryan` va maintenant exécuter le contrôleur `HelloController::indexAction()` et passer la valeur `ryan``en tant que variable ``$name` . Créer une « page » signifie simplement créer une méthode contrôleur et une route associée.

Notez la syntaxe utilisée pour faire référence au contrôleur : `AcmeHelloBundle:Hello:index`. Symfony2 utilise une notation de chaîne de caractères flexible pour faire référence aux différents contrôleurs. C'est la syntaxe la plus commune qui spécifie à Symfony2 de chercher une classe contrôleur appelée `HelloController` dans un bundle appelé `AcmeHelloBundle`. La méthode `indexAction()` est alors exécutée.

Pour plus de détails sur le format de chaîne de caractères utilisé pour référencer les différents contrôleurs, lisez *Pattern de Nommage du Contrôleur*.

Cet exemple place la configuration de routage directement dans le répertoire `app/config/`. Il existe une meilleure façon d'organiser vos routes : placer chacune d'entre elles dans le bundle auquel elles appartiennent. Pour plus d'informations, lisez *Inclure des Ressources Externes de Routage*.

Vous pouvez en apprendre beaucoup plus sur le système de routage en lisant le chapitre *Routage*.

Les paramètres de la route en tant qu'arguments du contrôleur

Vous savez déjà que le paramètre `AcmeHelloBundle:Hello:index` de `_controller` réfère à une méthode `HelloController::indexAction()` qui réside dans le bundle `AcmeHelloBundle`. Mais les arguments qui sont passés à cette méthode sont encore plus intéressants:

Listing 5-4
```php
// src/Acme/HelloBundle/Controller/HelloController.php
namespace Acme\HelloBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```
5
6  class HelloController extends Controller
7  {
8      public function indexAction($name)
9      {
10       // ...
11     }
12 }
```

Le contrôleur possède un argument unique, `$name`, qui correspond au paramètre `{name}` de la route associée (`ryan` dans notre exemple). En fait, lorsque vous exécutez votre contrôleur, Symfony2 fait correspondre chaque argument du contrôleur avec un paramètre de la route correspondante. Prenez l'exemple suivant :

Listing 5-5
```
1 # app/config/routing.yml
2 hello:
3     path:      /hello/{first_name}/{last_name}
4     defaults:  { _controller: AcmeHelloBundle:Hello:index, color: green }
```

Le contrôleur dans cet exemple peut prendre plusieurs arguments:

Listing 5-6
```
1 public function indexAction($first_name, $last_name, $color)
2 {
3     // ...
4 }
```

Notez que les deux variables de substitution (`{first_name}`, `{last_name}`) ainsi que la variable par défaut `color` sont disponibles en tant qu'arguments dans le contrôleur. Quand une route correspond, les variables de substitution sont fusionnées avec celles `par défaut` afin de construire un tableau qui est à la disposition de votre contrôleur.

Faire correspondre les paramètres de la route aux arguments du contrôleur est facile et flexible. Gardez les directives suivantes en tête quand vous développez.

- **L'ordre des arguments du contrôleur n'a pas d'importance**

  Symfony est capable de faire correspondre les noms des paramètres de la route aux noms des variables de la signature de la méthode du contrôleur. En d'autres termes, il réalise que le paramètre `{last_name}` correspond à l'argument `$last_name`. Les arguments du contrôleur pourraient être totalement réorganisés, cela fonctionnerait toujours parfaitement :

  Listing 5-7
  ```
  1 public function indexAction($last_name, $color, $first_name)
  2 {
  3     // ..
  4 }
  ```

- **Chaque argument du contrôleur doit correspondre à un paramètre de la route**

  Le code suivant lancerait une `RuntimeException` parce qu'il n'y a pas de paramètre `foo` défini dans la route :

  Listing 5-8
  ```
  1 public function indexAction($first_name, $last_name, $color, $foo)
  2 {
  ```

```
3       // ..
4   }
```

Cependant, définir l'argument en tant qu'optionnel est parfaitement valide. L'exemple suivant ne lancerait pas d'exception :

Listing 5-9
```
1  public function indexAction($first_name, $last_name, $color, $foo =
2  'bar')
3  {
4      // ..
   }
```

- **Tous les paramètres de la route n'ont pas besoin d'être des arguments de votre contrôleur**

  Si, par exemple, le paramètre `last_name` n'était pas important pour votre contrôleur, vous pourriez complètement l'omettre :

Listing 5-10
```
1  public function indexAction($first_name, $color)
2  {
3      // ..
4  }
```

Chaque route possède aussi un paramètre spécial `_route` qui est égal au nom de la route qui a été reconnue (par exemple: `hello`). Bien que généralement inutile , il est néanmoins disponible en tant qu'argument du contrôleur au même titre que les autres.

La `Requête` en tant qu'argument du Contrôleur

Pour plus de facilités, Symfony peut aussi vous passer l'objet `Request` en tant qu'argument de votre contrôleur. Ceci est spécialement pratique lorsque vous travaillez avec les formulaires, par exemple:

Listing 5-11
```
1  use Symfony\Component\HttpFoundation\Request;
2
3  public function updateAction(Request $request)
4  {
5      $form = $this->createForm(...);
6
7      $form->handleRequest($request);
8      // ...
9  }
```

Créer une page statique

Il est possible de créer une page web statique sans même créer de contrôleur (seuls un template et une route sont nécessaires).

N'hésitez pas à l'utiliser ! Rendez-vous sur la page de cookbook `/cookbook/templating/render_without_controller`.

La Classe Contrôleur de Base

Afin de vous faciliter le travail, Symfony2 est fourni avec une classe `Controller` de base qui vous assiste dans les tâches les plus communes et qui donne à votre classe contrôleur l'accès à n'importe quelle ressource dont elle pourrait avoir besoin. En étendant cette classe `Controller`, vous pouvez tirer parti de plusieurs méthodes utiles.

Ajoutez le mot-clé `use` au-dessus de la classe `Controller` et modifiez `HelloController` pour qu'il l'étende:

Listing 5-12

```
// src/Acme/HelloBundle/Controller/HelloController.php
namespace Acme\HelloBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class HelloController extends Controller
{
    public function indexAction($name)
    {
        return new Response('<html><body>Hello '.$name.'!</body></html>');
    }
}
```

Cela ne change en fait rien au fonctionnement de votre contrôleur. Dans la prochaine section, vous en apprendrez plus sur les méthodes d'aide (« helper ») que la classe contrôleur de base met à votre disposition. Ces méthodes sont juste des raccourcis pour utiliser des fonctionnalités coeurs de Symfony2 qui sont à votre disposition en utilisant ou non la classe `Controller` de base. Une bonne manière de se rendre compte de son efficacité est de regarder le code de la classe *`Controller`*[1] elle-même.

Étendre la classe de base est *facultatif* dans Symfony; elle contient des raccourcis utiles mais rien d'obligatoire. Vous pouvez aussi étendre *`ContainerAware`*[2]. L'objet conteneur de service (« service container ») sera ainsi accessible à travers la propriété `container`.

Vous pouvez aussi définir vos *Contrôleurs en tant que Services*. Ce n'est pas obligatoire mais cela vous permet de mieux contrôler les dépendances qui sont effectivement injectées dans votre contrôleur.

Les Tâches Communes du Contrôleur

Bien qu'un contrôleur puisse tout faire en théorie, la plupart d'entre-eux va accomplir les mêmes tâches basiques encore et toujours. Ces tâches, comme rediriger, forwarder, afficher des templates et accéder aux services sont très faciles à gérer dans Symfony2.

Rediriger

Si vous voulez rediriger l'utilisateur sur une autre page, utilisez la méthode `redirect()`:

Listing 5-13

1. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle/Controller/Controller.html
2. http://api.symfony.com/master/Symfony/Component/DependencyInjection/ContainerAware.html

```php
public function indexAction()
{
    return $this->redirect($this->generateUrl('homepage'));
}
```

La méthode `generateUrl()` est juste une fonction d'aide qui génère une URL pour une route donnée. Pour plus d'informations, lisez le chapitre *Routage*.

Par défaut, la méthode `redirect()` produit une redirection 302 (temporaire). Afin d'exécuter une redirection 301 (permanente), modifiez le second argument:

Listing 5-14
```php
public function indexAction()
{
    return $this->redirect($this->generateUrl('homepage'), 301);
}
```

La méthode `redirect()` est simplement un raccourci qui crée un objet `Response` spécialisé dans la redirection d'utilisateur. Cela revient à faire:

Listing 5-15
```php
use Symfony\Component\HttpFoundation\RedirectResponse;

return new RedirectResponse($this->generateUrl('homepage'));
```

Forwarder

Vous pouvez aussi facilement forwarder sur un autre contrôleur en interne avec la méthode `forward()`. Plutôt que de rediriger le navigateur de l'utilisateur, elle effectue une sous-requête interne, et appelle le contrôleur spécifié. La méthode `forward()` retourne l'objet `Response` qui est retourné par ce contrôleur:

Listing 5-16
```php
public function indexAction($name)
{
    $response = $this->forward('AcmeHelloBundle:Hello:fancy', array(
        'name'  => $name,
        'color' => 'green',
    ));

    // ... modifiez encore la réponse ou bien retournez-la directement

    return $response;
}
```

Notez que la méthode *forward()* utilise la même représentation de chaîne de caractères du contrôleur que celle utilisée dans la configuration de routage. Dans ce cas, la classe contrôleur cible va être `HelloController` dans le bundle `AcmeHelloBundle`. Le tableau passé à la méthode devient les arguments du contrôleur. Cette même interface est utilisée lorsque vous intégrez des contrôleurs dans des templates (voir *Contrôleurs imbriqués*). La méthode contrôleur cible devrait ressembler à quelque chose comme:

Listing 5-17
```php
public function fancyAction($name, $color)
{
    // ... crée et retourne un objet Response
}
```

Et comme quand vous créez un contrôleur pour une route, l'ordre des arguments de `fancyAction` n'a pas d'importance. Symfony2 fait correspondre le nom des clés d'index (par exemple: `name`) avec le nom des arguments de la méthode (par exemple: `$name`). Si vous changez l'ordre des arguments, Symfony2 va toujours passer la valeur correcte à chaque variable.

Comme d'autres méthodes de base de `Controller`, la méthode `forward` est juste un raccourci vers une fonctionnalité coeur de Symfony2. Un forward peut être exécuté directement via le service `http_kernel` et retourne un objet `Response` :

Listing 5-18
```php
$httpKernel = $this->container->get('http_kernel');
$response = $httpKernel->forward('AcmeHelloBundle:Hello:fancy', array(
    'name'  => $name,
    'color' => 'green',
));
```

Afficher des Templates

Bien que ce n'est pas obligatoire, la plupart des contrôleurs va finalement retourner un template qui sera chargé de générer du HTML (ou un autre format) pour le contrôleur. La méthode `renderView()` retourne un template et affiche son contenu. Le contenu du template peut être utilisé pour créer un objet `Response`:

Listing 5-19
```php
use Symfony\Component\HttpFoundation\Response;

$content = $this->renderView(
    'AcmeHelloBundle:Hello:index.html.twig',
    array('name' => $name)
);

return new Response($content);
```

Cela peut même être effectué en une seule étape à l'aide de la méthode `render()`, qui retourne un objet `Response` contenant le contenu du template:

Listing 5-20
```php
return $this->render(
    'AcmeHelloBundle:Hello:index.html.twig',
    array('name' => $name)
);
```

Dans les deux cas, le template `Resources/views/Hello/index.html.twig` dans `AcmeHelloBundle` sera affiché.

Le moteur de rendu (« templating engine ») de Symfony est expliqué plus en détail dans le chapitre *Templating*

Vous pouvez même éviter d'appeler la méthode `render` en utilisant l'annotation `@Template`. Lisez la documentation du *FrameworkExtraBundle* pour plus de détails.

La méthode `renderView` est un raccourci vers l'utilisation directe du service `templating`. Ce dernier peut aussi être utilisé directement:

Listing 5-21
```php
$templating = $this->get('templating');
$content = $templating->render(
    'AcmeHelloBundle:Hello:index.html.twig',
    array('name' => $name)
);
```

Il est aussi possible d'afficher des templates situés dans des sous-répertoires. Mais évitez tout de même de tomber dans la facilité de faire des arborescences trop élaborées:

Listing 5-22
```php
$templating->render(
    'AcmeHelloBundle:Hello/Greetings:index.html.twig',
    array('name' => $name)
);
// Affiche index.html.twig situé dans Resources/views/Hello/Greetings.
```

Accéder à d'autres Services

Quand vous étendez la classe contrôleur de base, vous pouvez utiliser n'importe quel service Symfony2 via la méthode `get()`. Voici plusieurs services communs dont vous pourriez avoir besoin:

Listing 5-23
```php
$request = $this->getRequest();

$templating = $this->get('templating');

$router = $this->get('router');

$mailer = $this->get('mailer');
```

Il y a d'innombrables autres services à votre disposition et vous êtes encouragé à définir les vôtres. Pour lister tous les services disponibles, utilisez la commande de la console `container:debug` :

Listing 5-24
```
$ php app/console container:debug
```

Pour plus d'informations, voir le chapitre *Service Container*.

Gérer les Erreurs et les Pages 404

Quand « quelque chose » n'est pas trouvé, vous devriez vous servir correctement du protocole HTTP et retourner une réponse 404. Pour ce faire, vous allez lancer un type spécial d'exception. Si vous étendez la classe contrôleur de base, faites comme ç:

Listing 5-25
```php
public function indexAction()
{
    // récupérer l'objet depuis la base de données
    $product = ...;
    if (!$product) {
        throw $this->createNotFoundException('Le produit n\'existe pas');
```

```
    }

    return $this->render(...);
}
```

La méthode `createNotFoundException()` crée un objet spécial `NotFoundHttpException`, qui finalement déclenche une réponse HTTP 404 dans Symfony.

Évidemment, vous êtes libre de lever n'importe quelle `Exception` dans votre contrôleur - Symfony2 retournera automatiquement un code de réponse HTTP 500.

Listing 5-26
```
throw new \Exception('Quelque chose a mal tourné!');
```

Dans chaque cas, une page d'erreur avec style est retournée à l'utilisateur final et une page d'erreur complète avec des infos de debugging est retournée au développeur (lorsqu'il affiche cette page en mode debug). Ces deux pages d'erreur peuvent être personnalisées. Pour de plus amples détails, lisez la partie du cookbook « *Comment personnaliser les pages d'erreur* ».

Gérer la Session

Symfony2 fournit un objet session sympa que vous pouvez utiliser pour stocker de l'information à propos de l'utilisateur (que ce soit une personne réelle utilisant un navigateur, un bot, ou un service web) entre les requêtes. Par défaut, Symfony2 stocke les attributs dans un cookie en utilisant les sessions natives de PHP.

Stocker et récupérer des informations depuis la session peut être effectué facilement depuis n'importe quel contrôleur:

Listing 5-27
```
$session = $this->getRequest()->getSession();

// stocke un attribut pour une réutilisation lors d'une future requête utilisateur
$session->set('foo', 'bar');

// dans un autre contrôleur pour une autre requête
$foo = $session->get('foo');

// utilise une valeur par défaut si la clé n'existe pas
$filters = $session->get('filters', array());
```

Ces attributs vont rester affectés à cet utilisateur pour le restant de son temps session.

Les Messages Flash

Vous pouvez aussi stocker de petits messages qui vont être gardés dans la session de l'utilisateur pour la requête suivante uniquement. Ceci est utile lors du traitement d'un formulaire : vous souhaitez rediriger l'utilisateur et afficher un message spécial lors de la *prochaine* requête. Ces types de message sont appelés messages « flash ».

Par exemple, imaginez que vous traitiez la soumission d'un formulaire:

Listing 5-28
```
public function updateAction()
{
    $form = $this->createForm(...);

```

```
    $form->handleRequest($this->getRequest());

    if ($form->isValid()) {
        // effectue le traitement du formulaire

        $this->get('session')->getFlashBag()->add(
            'notice',
            'Vos changements ont été sauvegardés!'
        );

        return $this->redirect($this->generateUrl(...));
    }

    return $this->render(...);
}
```

Après avoir traité la requête, le contrôleur définit un message flash `notice` et puis redirige l'utilisateur. Le nom (`notice`) n'est pas très important - c'est juste ce que vous utilisez pour identifier le type du message.

Dans le template de la prochaine action, le code suivant pourra être utilisé pour afficher le message `notice` :

Listing 5-29
```
{% for flashMessage in app.session.flashbag.get('notice') %}
    <div class="flash-notice">
        {{ flashMessage }}
    </div>
{% endfor %}
```

De par leur conception, les messages flash sont faits pour durer pendant exactement une requête (ils « disparaissent en un éclair/flash »). Ils sont conçus pour être utilisés avec les redirections exactement comme vous l'avez fait dans cet exemple.

L'Objet Response

La seule condition requise d'un contrôleur est de retourner un objet `Response`. La classe *`Response`*[3] est une abstraction PHP autour de la réponse HTTP - le message texte est complété avec des en-têtes HTTP et du contenu qui est envoyé au client:

Listing 5-30
```
use Symfony\Component\HttpFoundation\Response;

// crée une simple Réponse avec un code de statut 200 (celui par défaut)
$response = new Response('Hello '.$name, 200);

// crée une réponse JSON avec un code de statut 200
$response = new Response(json_encode(array('name' => $name)));
$response->headers->set('Content-Type', 'application/json');
```

La propriété `headers` est un objet *`HeaderBag`*[4] avec plusieurs méthodes utiles pour lire et transformer les en-têtes de la `Response`. Les noms des en-têtes sont normalisés et ainsi, utiliser `Content-Type` est équivalent à `content-type` ou même `content_type`.

3. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html
4. http://api.symfony.com/master/Symfony/Component/HttpFoundation/HeaderBag.html

Il existe des classes spéciales pour construire des réponses plus facilement:

- Pour du JSON, *`JsonResponse`*[5]. Lisez *component-http-foundation-json-response*.
- Pour des fichiers, *`BinaryFileResponse`*[6]. Lisez *component-http-foundation-serving-files*.

L'Objet Request

En plus des paramètres de routes, le contrôleur a aussi accès à l'objet `Request` quand il étend la classe `Controller` de base:

Listing 5-31
```php
$request = $this->getRequest();

$request->isXmlHttpRequest(); // est-ce une requête Ajax?

$request->getPreferredLanguage(array('en', 'fr'));

$request->query->get('page'); // retourne un paramètre $_GET

$request->request->get('page'); // retourne un paramètre $_POST
```

Comme l'objet `Response`, les en-têtes de la requête sont stockées dans un objet `HeaderBag` et sont facilement accessibles.

Le mot de la fin

Chaque fois que vous créez une page, vous allez au final avoir besoin d'écrire du code qui contient la logique de cette page. Dans Symfony, ceci est appelé un contrôleur, et c'est une fonction PHP qui peut faire tout ce qu'il faut pour retourner l'objet final `Response` qui sera délivré à l'utilisateur.

Pour vous simplifier la vie, vous pouvez choisir d'étendre une classe `Controller` de base, qui contient des méthodes raccourcies pour de nombreuses tâches communes d'un contrôleur. Par exemple, sachant que vous ne voulez pas mettre de code HTML dans votre contrôleur, vous pouvez utiliser la méthode `render()` pour délivrer et retourner le contenu d'un template.

Dans d'autres chapitres, vous verrez comment le contrôleur peut être utilisé pour sauvegarder et aller chercher des objets dans une base de données, traiter des soumissions de formulaires, gérer le cache et plus encore.

En savoir plus grâce au Cookbook

- *Comment personnaliser les pages d'erreur*
- *Comment définir des contrôleurs en tant que Services*

5. http://api.symfony.com/master/Symfony/Component/HttpFoundation/JsonResponse.html
6. http://api.symfony.com/master/Symfony/Component/HttpFoundation/BinaryFileResponse.html

Chapter 6
Routage

De belles URLs sont une obligation absolue pour une quelconque application web sérieuse. Cela implique d'oublier les URLs moches comme `index.php?article_id=57` en faveur de quelque chose comme `/read/intro-to-symfony`.

Avoir de la flexibilité est encore plus important. Que se passe-t-il si vous avez besoin de changer l'URL d'une page de `/blog` pour `/news` ? Combien de liens aurez-vous besoin de traquer et mettre à jour afin de prendre en compte ce changement ? Si vous utilisez le routeur de Symfony, le changement est simple.

Le routeur Symfony2 vous laisse définir des URLs créatives que vous faites correspondre à différents points de votre application. A la fin de ce chapitre, vous serez capable de :

- Créer des routes complexes qui correspondent à des contrôleurs
- Générer des URLs à l'intérieur des templates et des contrôleurs
- Charger des ressources de routage depuis des bundles (ou depuis ailleurs)
- Débugger vos routes

Le routage en Action

Une *route* est une correspondance entre un pattern d'URL et un contrôleur. Par exemple, supposez que vous vouliez faire correspondre n'importe quelle URL comme `/blog/my-post` ou `/blog/all-about-symfony` et l'envoyer vers un contrôleur qui puisse chercher et retourner ce billet de blog. La route est simple :

Listing 6-1
```yaml
# app/config/routing.yml
blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

Le pattern défini par la route `blog_show` agit comme `/blog/*` où le joker (l'étoile) possède le nom `slug`. Pour l'URL `/blog/my-blog-post`, la variable `slug` prend la valeur de `my-blog-post`, qui est à votre disposition dans votre contrôleur (continuez à lire).

Le paramètre `_controller` est une clé spéciale qui dit à Symfony quel contrôleur doit être exécuté lorsqu'une URL correspond à cette route. La chaîne de caractères `_controller` est appelée le *nom logique*. Il suit un pattern qui pointe vers une classe et une méthode PHP spécifique:

Listing 6-2
```php
// src/Acme/BlogBundle/Controller/BlogController.php
namespace Acme\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($slug)
    {
        $blog = // utilisez la variable $slug pour interroger la base de données

        return $this->render('AcmeBlogBundle:Blog:show.html.twig', array(
            'blog' => $blog,
        ));
    }
}
```

Félicitations ! Vous venez juste de créer votre première route et de la connecter à un contrôleur. Maintenant, quand vous visitez `/blog/my-post`, le contrôleur `showAction` va être exécuté et la variable `$slug` aura la valeur `my-post`.

Ceci est le but du routeur Symfony2 : faire correspondre l'URL d'une requête à un contrôleur. Tout au long du chemin, vous allez apprendre toutes sortes d'astuces qui rendent même facile la création des URLs les plus complexes.

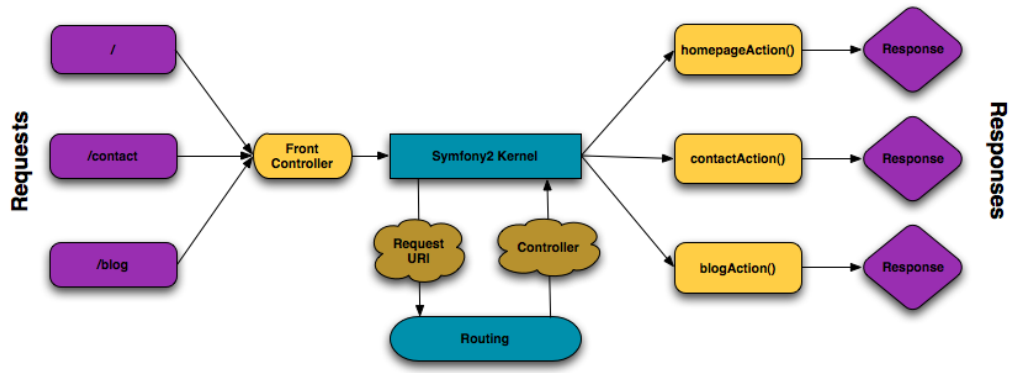
Routage: Sous le Capot

Quand une requête est lancée vers votre application, elle contient une adresse pointant sur la « ressource » exacte que le client désire. Cette adresse est appelée l'URL, (ou l'URI), et pourrait être `/contact`, `/blog/read-me`, ou n'importe quoi d'autre. Prenez l'exemple de la requête HTTP suivante :

Listing 6-3
```
GET /blog/my-blog-post
```

Le but du système de routage de Symfony2 est d'analyser cette URL et de déterminer quel contrôleur devrait être exécuté. Le déroulement complet ressemble à ça :

1. La requête est gérée par le contrôleur frontal de Symfony2 (par exemple : `app.php`) ;
2. Le coeur de Symfony2 (c-a-d Kernel) demande au routeur d'inspecter la requête ;
3. Le routeur fait correspondre l'URL entrante à une route spécifique et retourne l'information à propos de la route, incluant le contrôleur qui devrait être exécuté ;
4. Le Kernel Symfony2 exécute le contrôleur, qui finalement retourne un objet `Response`.



La partie routage est un outil qui traduit l'URL entrante en un contrôleur spécifique à exécuter.

Créer des Routes

Symfony charge toutes les routes de votre application depuis un fichier unique de configuration du routage. Le fichier est généralement localisé dans le répertoire `app/config/routing.yml`, mais peut être configuré afin d'être n'importe quoi d'autre (incluant un fichier XML ou PHP) via le fichier de configuration de l'application :

Listing 6-4
```yaml
# app/config/config.yml
framework:
    # ...
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
```

Bien que toutes les routes soient chargées depuis un fichier unique, c'est une pratique courante d'inclure des ressources de routage additionnelles. Pour faire cela, il vous suffit de spécifier quels fichiers externes doivent être inclus dans le fichier de configuration de routage principal.Référez-vous à la section *Inclure des Ressources Externes de Routage* pour plus d'informations.

Configuration Basique des Routes

Définir une route est facile, et une application typique va avoir de nombreuses routes. Une route basique consiste simplement de deux parties : le `pattern` à comparer et un tableau `defaults` :

Listing 6-5
```yaml
_welcome:
    pattern:   /
    defaults:  { _controller: AcmeDemoBundle:Main:homepage }
```

Cette route correspond à la page d'accueil (`/`) et la relie avec le contrôleur `AcmeDemoBundle:Main:homepage`. La chaîne de caractères `_controller` est traduite par Symfony2 en une fonction PHP qui est ensuite exécutée. Ce processus sera expliqué rapidement dans la section *Pattern de Nommage du Contrôleur*.

Routage avec les Paramètres de substitution

Évidemment, le système de routage supporte des routes beaucoup plus intéressantes. De nombreuses routes vont contenir un ou plusieurs paramètres de substitution nommés « joker » :

Listing 6-6
```yaml
blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

Le pattern va faire correspondre tout ce qui ressemble à `/blog/*`. Mieux encore, la valeur correspondante au paramètre de substitution `{slug}` sera disponible dans votre contrôleur. En d'autres termes, si l'URL est `/blog/hello-world`, une variable `$slug`, avec la valeur `hello-world`, sera à votre disposition dans le contrôleur. Ceci peut être utilisé, par exemple, pour récupérer l'entrée du blog qui correspond à cette chaîne de caractères.

Cependant, le pattern *ne va pas* faire correspondre simplement `/blog` à cette route. Et cela parce que, par défaut, tous les paramètres de substitution sont requis. Ce comportement peut être modifié en ajoutant une valeur au paramètre de substitution dans le tableau `defaults`.

Paramètres de substitution Requis et Optionnels

Pour rendre les choses plus excitantes, ajoutez une nouvelle route qui affiche une liste de toutes les entrées du blog disponibles pour cette application imaginaire :

Listing 6-7
```
1 blog:
2     pattern:   /blog
3     defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

Jusqu'ici, cette route est aussi simple que possible - elle ne contient pas de paramètres de substitution et va correspondre uniquement à l'URL exacte `/blog`. Mais que se passe-t-il si vous avez besoin que cette route supporte la pagination, où `blog/2` affiche la seconde page des entrées du blog ? Mettez la route à jour afin qu'elle ait un nouveau paramètre de substitution `{page}` :

Listing 6-8
```
1 blog:
2     pattern:   /blog/{page}
3     defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

Comme le paramètre de substitution `{slug}` d'avant, la valeur correspondante à `{page}` va être disponible dans votre contrôleur. Cette dernière peut être utilisée pour déterminer quel ensemble d'entrées blog doit être délivré pour la page donnée.

Mais attendez ! Sachant que les paramètres substitutifs sont requis par défaut, cette route va maintenant arrêter de correspondre à une requête contenant simplement l'URL `/blog`. A la place, pour voir la page 1 du blog, vous devriez utiliser l'URL `/blog/1` ! Ceci n'étant pas une solution « viable » pour une telle application web, modifiez la route et faites en sorte que le paramètre `{page}` soit optionnel. Vous pouvez faire cela en l'incluant dans la collection des `defaults` :

Listing 6-9
```
1 blog:
2     pattern:   /blog/{page}
3     defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }
```

En ajoutant `page` aux clés `defaults`, le paramètre substitutif `{page}` n'est donc plus obligatoire. L'URL `/blog` va correspondre à cette route et la valeur du paramètre `page` sera défini comme étant `1`. L'URL `/blog/2` quant à elle va aussi correspondre, donnant au paramètre `page` la valeur `2`. Parfait.

/blog	{page} = 1
/blog/1	{page} = 1
/blog/2	{page} = 2

Les routes avec des paramètres optionnels à la fin ne correspondront pas à la requête demandée s'il y a un slash à la fin (ex `/blog/` ne correspondra pas, `/blog` correspondra).

Ajouter des Conditions Requises

Regardez rapidement les routes qui ont été créées jusqu'ici :

Listing 6-10
```
1 blog:
2     pattern:   /blog/{page}
3     defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }
4
```

```
5  blog_show:
6      pattern:   /blog/{slug}
7      defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

Pouvez-vous voir le problème ? Notez que les deux routes possèdent des patterns qui correspondent aux URLs ressemblant à `/blog/*`. Le routeur Symfony choisira toujours la **première** route correspondante qu'il trouve. En d'autres mots, la route `blog_show` ne sera *jamais* utilisée. Néanmoins, une URL comme `/blog/my-blog-post` correspondra à la première route (`blog`) qui retournera une valeur (n'ayant aucun sens) `my-blog-post` au paramètre `{page}`.

URL	route	paramètres
/blog/2	blog	{page} = 2
/blog/my-blog-post	blog	{page} = my-blog-post

La réponse au problème est d'ajouter des conditions requises à la route. Les routes de l'exemple ci-dessus fonctionneraient parfaitement si le pattern `/blog/{page}` correspondait *uniquement* aux URLs avec la partie `{page}` étant un nombre entier. Heureusement, des conditions requises sous forme d'expression régulière peuvent être facilement ajoutées pour chaque paramètre. Par exemple :

Listing 6-11

```
1  blog:
2      pattern:   /blog/{page}
3      defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }
4      requirements:
5          page:  \d+
```

La condition requise `\d+` est une expression régulière qui oblige la valeur du paramètre `{page}` à être un nombre (composé d'un ou plusieurs chiffres). La route `blog` correspondra toujours à une URL comme `/blog/2` (parce que 2 est un nombre), mais elle ne correspondra par contre plus à une URL comme `/blog/my-blog-post` (car `my-blog-post` n'est *pas* un nombre).

Comme résultat, une URL comme `/blog/my-blog-post` va dorénavant correspondre correctement à la route `blog_show`.

URL	route	paramètres
/blog/2	blog	{page} = 2
/blog/my-blog-post	blog_show	{slug} = my-blog-post

Les Routes précédentes Gagnent toujours

Tout cela signifie que l'ordre des routes est très important. Si la route `blog_show` était placée au-dessus de la route `blog`, l'URL `/blog/2` correspondrait à `blog_show` au lieu de `blog` puisque le paramètre {slug}`` de `blog_show` n'a pas de conditions requises. En utilisant un ordre clair et intelligent, vous pouvez accomplir tout ce que vous voulez.

Avec la possibilité de définir des conditions requises pour les paramètres à l'aide d'expressions régulières, la complexité et la flexibilité de chaque condition est entièrement dépendante de ce que vous en faites. Supposez que la page d'accueil de votre application soit disponible en deux langues différentes, basée sur l'URL :

Listing 6-12

```yaml
homepage:
    pattern:   /{culture}
    defaults:  { _controller: AcmeDemoBundle:Main:homepage, culture: en }
    requirements:
        culture:  en|fr
```

Pour les requêtes entrantes, la partie `{culture}` de l'URL est comparée à l'expression régulière `(en|fr)`.

/	{culture} = en
/en	{culture} = en
/fr	{culture} = fr
/es	*ne correspondra pas à cette route*

Ajouter des Conditions Requises pour la Méthode HTTP

En plus de l'URL, vous pouvez aussi comparer la *méthode* (c-a-d GET, HEAD, POST, PUT, DELETE) de la requête entrante avec celle définie dans les conditions requises de la route. Supposez que vous ayez un formulaire de contact avec deux contrôleurs - un pour afficher le formulaire (quand on a une requête GET) et un pour traiter le formulaire une fois qu'il a été soumis (avec une requête POST). Ceci peut être accompli avec la configuration de routage suivante :

Listing 6-13
```yaml
contact:
    pattern:  /contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
    requirements:
        _method:  GET

contact_process:
    pattern:  /contact
    defaults: { _controller: AcmeDemoBundle:Main:contactProcess }
    requirements:
        _method:  POST
```

Malgré le fait que ces deux routes aient des patterns identiques (`/contact`), la première route correspondra uniquement aux requêtes GET et la seconde route correspondra seulement aux requêtes POST. Cela signifie que vous pouvez afficher le formulaire et le soumettre via la même URL, tout en utilisant des contrôleurs distincts pour les deux actions.

Si aucune condition requise `_method` n'est spécifiée, la route correspondra à *toutes* les méthodes.

Comme les autres, la condition requise `_method` est analysée en tant qu'expression régulière. Pour faire correspondre les requêtes à la méthode `GET` *ou* à `POST`, vous pouvez utiliser `GET|POST`.

Ajouter un Pattern « Hostname » (« nom d'hôte » en français)

New in version 2.2.

Vous pouvez aussi faire la correspondance avec le *hostname* (« nom d'hôte » en français) HTTP de la requête entrante :

Listing 6-14
```yaml
mobile_homepage:
    pattern:  /
    hostname_pattern: m.example.com
    defaults: { _controller: AcmeDemoBundle:Main:mobileHomepage }

homepage:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

Les deux routes correspondent au même pattern `/`, cependant la première va correspondre uniquement si le nom d'hôte est `m.example.com`.

Valeurs de substitution et Conditions Requises pour les Patterns Hostname (« nom d'hôte » en français)

Des valeurs de substitution peuvent être utilisées dans les patterns de noms d'hôte ainsi que dans les patterns, et les conditions requises s'appliquent aussi à elles.

Dans l'exemple suivant, nous évitons de coder en dur (« hardcoding » en anglais) le nom de domaine en utilisant une valeur de substitution et une condition requise. `%domain%` dans les conditions requises est remplacé par la valeur du paramètre `domain` venant du conteneur d'injection de dépendances.

Listing 6-15
```yaml
mobile_homepage:
    pattern:  /
    hostname_pattern: m.{domain}
    defaults: { _controller: AcmeDemoBundle:Main:mobileHomepage }
    requirements:
        domain: %domain%

homepage:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Main:homepage }
```

Exemple de Routage Avancé

A ce point, vous connaissez tout ce dont vous avez besoin pour créer une structure de routage puissante dans Symfony. Ce qui suit est un exemple montrant simplement à quel point le système de routage peut être flexible :

Listing 6-16
```yaml
article_show:
  pattern:  /articles/{culture}/{year}/{title}.{_format}
  defaults: { _controller: AcmeDemoBundle:Article:show, _format: html }
  requirements:
      culture:  en|fr
      _format:  html|rss
      year:     \d+
```

Comme vous l'avez vu, cette route correspondra uniquement si la partie `{culture}` de l'URL est `en` ou `fr` et si `{year}` est un nombre. Cette route montre aussi comment vous pouvez utiliser un point entre les paramètres de substitution à la place d'un slash. Les URLs qui correspondent à cette route pourraient ressembler à ça :

- `/articles/en/2010/my-post`
- `/articles/fr/2010/my-post.rss`

Le Paramètre Spécial de Routage `_format`

Cet exemple met aussi en valeur le paramètre spécial de routage `_format`. Lorsque vous utilisez ce paramètre, la valeur correspondante devient alors le « format de la requête » de l'objet `Request`. Finalement, le format de la requête est utilisé pour des taches comme spécifier le `Content-Type` de la réponse (par exemple : un format de requête `json` se traduit en un `Content-Type` ayant pour valeur `application/json`). Il peut aussi être utilisé dans le contrôleur pour délivrer un template différent pour chaque valeur de `_format`. Le paramètre `_format` est une manière très puissante de délivrer le même contenu dans différents formats.

Paramètres spéciaux de routing

Comme vous l'avez vu, chaque paramètre ou valeur par défaut peut être passé comme argument à la méthode contrôleur. De plus, il y a 3 paramètres spéciaux : chacun d'eux apporte une fonctionnalité unique à votre application :

- `_controller`: Comme vous l'avez vu, ce paramètre est utilisé pour déterminer quel contrôleur est exécuté lorsque l'URL est reconnue ;
- `_format`: Utilisé pour définir le format de la requête(*en savoir plus*) ;
- `_locale`: Utilisé pour définir la locale de la session (*en savoir plus*).

Si vous utilisez le paramètre `_locale` dans une route, cette valeur sera également stockée en session pour que les futures requêtes la conservent.

Pattern de Nommage du Contrôleur

Chaque route doit avoir un paramètre `_controller`, qui lui dicte quel contrôleur doit être exécuté lorsque cette route correspond à l'URL. Ce paramètre utilise un pattern de chaîne de caractères simple appelé *nom logique du contrôleur*, que Symfony fait correspondre à une méthode et à une classe PHP spécifique. Le pattern a trois parties, chacune séparée par deux-points :

bundle:**contrôleur**:**action**

Par exemple, la valeur `AcmeBlogBundle:Blog:show` pour le paramètre `_controller` signifie :

Bundle	Classe du Contrôleur	Nom de la Méthode
AcmeBlogBundle	BlogController	showAction

Le contrôleur pourrait ressembler à quelque chose comme ça:

Listing 6-17
```php
// src/Acme/BlogBundle/Controller/BlogController.php
namespace Acme\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
```

```
7  {
8      public function showAction($slug)
9      {
10         // ...
11     }
12 }
```

Notez que Symfony ajoute la chaîne de caractères `Controller` au nom de la classe (`Blog` => `BlogController`) et `Action` au nom de la méthode (`show` => `showAction`).

Vous pourriez aussi faire référence à ce contrôleur en utilisant le nom complet de sa classe et de sa méthode : `Acme\BlogBundle\Controller\BlogController::showAction`. Mais si vous suivez quelques conventions simples, le nom logique est plus concis et permet aussi plus de flexibilité.

En plus d'utiliser le nom logique ou le nom complet de la classe, Symfony supporte une troisième manière de référer à un contrôleur. Cette méthode utilise un seul séparateur deux-points (par exemple : `service_name:indexAction`) et réfère au contrôleur en tant que service (see *Comment définir des contrôleurs en tant que Services*).

Les Paramètres de la Route et les Arguments du Contrôleur

Les paramètres de la route (par exemple : `{slug}`) sont spécialement importants parce que chacun d'entre eux est mis à disposition en tant qu'argument de la méthode contrôleur:

Listing 6-18
```
1 public function showAction($slug)
2 {
3   // ...
4 }
```

En réalité, la collection entière `defaults` est fusionnée avec les valeurs des paramètres afin de former un unique tableau. Chaque clé du tableau est disponible en tant qu'argument dans le contrôleur.

En d'autres termes, pour chaque argument de votre méthode contrôleur, Symfony recherche un paramètre de la route avec ce nom et assigne sa valeur à cet argument. Dans l'exemple avancé ci-dessus, n'importe quelle combinaison (dans n'importe quel ordre) des variable suivantes pourrait être utilisée en tant qu'arguments de la méthode `showAction()` :

- `$culture`
- `$year`
- `$title`
- `$_format`
- `$_controller`

Sachant que les paramètres de substitution et la collection `defaults` sont fusionnés ensemble, même la variable `$_controller` est disponible. Pour une discussion plus détaillée sur le sujet, lisez *Les paramètres de la route en tant qu'arguments du contrôleur*.

Vous pouvez aussi utiliser une variable `$_route` spéciale, qui est définie comme étant le nom de la route qui a correspondu.

Inclure des Ressources Externes de Routage

Toutes les routes sont chargées via un unique fichier de configuration - généralement `app/config/routing.yml` (voir Créer des Routes ci-dessus). Cependant, la plupart du temps, vous voudrez charger les routes depuis d'autres endroits, comme un fichier de routage qui se trouve dans un bundle. Ceci peut être fait en « important » ce fichier :

Listing 6-19
```yaml
# app/config/routing.yml
acme_hello:
    resource: "@AcmeHelloBundle/Resources/config/routing.yml"
```

Lorsque vous importez des ressources depuis YAML, la clé (par exemple : `acme_hello`) n'a pas de sens précis. Assurez-vous simplement que cette dernière soit unique afin qu'aucune autre ligne ne la surcharge.

La clé de la `ressource` charge la ressource de routage donnée. Dans cet exemple, la ressource est le répertoire entier d'un fichier, où la syntaxe raccourcie `@AcmeHelloBundle` est traduite par le répertoire de ce bundle. Le fichier importé pourrait ressembler à quelque chose comme ça :

Listing 6-20
```yaml
 # src/Acme/HelloBundle/Resources/config/routing.yml
acme_hello:
     pattern:  /hello/{name}
     defaults: { _controller: AcmeHelloBundle:Hello:index }
```

Les routes de ce fichier sont analysées et chargées de la même manière que pour le fichier de routage principal.

Préfixer les Routes Importées

Vous pouvez aussi choisir de définir un « préfixe » pour les routes importées. Par exemple, supposez que vous vouliez que la route `acme_hello` ait un pattern final `/admin/hello/{name}` à la place de simplement `/hello/{name}` :

Listing 6-21
```yaml
# app/config/routing.yml
acme_hello:
    resource: "@AcmeHelloBundle/Resources/config/routing.yml"
    prefix:   /admin
```

La chaîne de caractères `/admin` sera maintenant ajoutée devant le pattern de chaque route chargée depuis la nouvelle ressource de routage.

Ajouter un Pattern Hostname (« nom d'hôte » en français) aux Routes Importées

New in version 2.2.

Vous pouvez définir un pattern de nom d'hôte pour les routes importées :

Listing 6-22
```yaml
# app/config/routing.yml
acme_hello:
```

```
3       resource: "@AcmeHelloBundle/Resources/config/routing.yml"
4       hostname_pattern: "hello.example.com"
```

Le pattern de nom d'hôte `hello.example.com` sera défini pour chaque route chargée depuis la nouvelle ressource de routage.

Vous pouvez également définir les routes en utilisant les annotations. Lisez la:doc:*documentation du FrameworkExtraBundle</bundles/SensioFrameworkExtraBundle/annotations/routing>* pour savoir comment faire.

Visualiser et Debugger les Routes

Lorsque vous ajoutez et personnalisez des routes, cela aide beaucoup de pouvoir visualiser et d'avoir des informations détaillées à propos de ces dernières. Une manière géniale de voir chaque route de votre application est d'utiliser la commande de la console `router:debug`. Exécutez la commande suivante depuis la racine de votre projet.

Listing 6-23
```
1  $ php app/console router:debug
```

La commande va retourner une liste de *toutes* les routes configurées dans votre application :

Listing 6-24
```
1  homepage             ANY       /
2  contact              GET       /contact
3  contact_process      POST      /contact
4  article_show         ANY       /articles/{culture}/{year}/{title}.{_format}
5  blog                 ANY       /blog/{page}
6  blog_show            ANY       /blog/{slug}
```

Vous pouvez aussi avoir des informations très spécifiques pour une seule route en incluant le nom de cette dernière après la commande :

Listing 6-25
```
1  $ php app/console router:debug article_show
```

New in version 2.1: La commande `router:match` a été ajoutée dans Symfony 2.1

Vous pouvez vérifier quelle route, s'il y en a, correspond à un chemin avec la commande `router:match` :

Listing 6-26
```
1  $ php app/console router:match /articles/en/2012/article.rss
2  Route "article_show" matches
```

Générer des URLs

Le système de routage devrait aussi être utilisé pour générer des URLs. En réalité, le routage est un système bidirectionnel : faire correspondre une URL à un contrôleur+paramètres et une

route+paramètres à une URL. Les méthodes *match()*[1] et *generate()*[2] forment ce système bi-directionnel. Prenez l'exemple de la route `blog_show` vue plus haut:

Listing 6-27
```php
$params = $router->match('/blog/my-blog-post');
// array('slug' => 'my-blog-post', '_controller' => 'AcmeBlogBundle:Blog:show')

$uri = $router->generate('blog_show', array('slug' => 'my-blog-post'));
// /blog/my-blog-post
```

Pour générer une URL, vous avez besoin de spécifier le nom de la route (par exemple : `blog_show`) ainsi que quelconque joker (par exemple : `slug = my-blog-post`) utilisé dans le pattern de cette route. Avec cette information, n'importe quelle URL peut être générée facilement:

Listing 6-28
```php
class MainController extends Controller
{
    public function showAction($slug)
    {
      // ...

      $url = $this->get('router')->generate('blog_show', array('slug' => 'my-blog-post'));
    }
}
```

Dans une prochaine section, vous apprendrez comment générer des URLs directement depuis les templates.

Si le front de votre application utilise des requêtes AJAX, vous voudrez sûrement être capable de générer des URLs en javascript en vous basant sur votre configuration. En utilisant le *FOSJsRoutingBundle*[3], vous pourrez faire cela :

Listing 6-29
```javascript
var url = Routing.generate('blog_show', { "slug": 'my-blog-post'});
```

Pour plus d'informations, lisez la documentation du bundle.

Générer des URLs Absolues

Par défaut, le routeur va générer des URLs relatives (par exemple : `/blog`). Pour générer une URL absolue, passez simplement `true` comme troisième argument de la méthode `generate()`:

Listing 6-30
```php
$router->generate('blog_show', array('slug' => 'my-blog-post'), true);
// http://www.example.com/blog/my-blog-post
```

L'host qui est utilisé lorsque vous générez une URL absolue est celui de l'objet courant `Request`. Celui-ci est détecté automatiquement basé sur les informations du serveur fournies par PHP. Lorsque vous générez des URLs absolues pour des scripts exécutés depuis la ligne de commande, vous devrez spécifier manuellement l'host désiré sur l'objet `RequestContext`:

Listing 6-31
```php
$router->getContext()->setHost('www.example.com');
```

1. http://api.symfony.com/master/Symfony/Component/Routing/Router.html#match()
2. http://api.symfony.com/master/Symfony/Component/Routing/Router.html#generate()
3. https://github.com/FriendsOfSymfony/FOSJsRoutingBundle

Générer des URLs avec « Query Strings »

La méthode `generate` prend un tableau de valeurs jokers pour générer l'URI. Mais si vous en passez d'autres, elles seront ajoutées à l'URI en tant que « query string »

Listing 6-32
```
$router->generate('blog', array('page' => 2, 'category' => 'Symfony'));
// /blog/2?category=Symfony
```

Générer des URLs depuis un template

L'endroit principal où vous générez une URL est depuis un template lorsque vous créez des liens entre les pages de votre application. Cela se fait comme on l'a vu auparavant, mais en utilisant une fonction d'aide (helper) du template :

Listing 6-33
```
<a href="{{ path('blog_show', { 'slug': 'my-blog-post' }) }}">
  Lire cette entrée blog.
</a>
```

Des URLs absolues peuvent aussi être générées.

Listing 6-34
```
<a href="{{ url('blog_show', { 'slug': 'my-blog-post' }) }}">
    Lire cette entrée blog.
</a>
```

Résumé

Le routage est un système permettant de faire correspondre l'URL des requêtes entrantes à une fonction contrôleur qui devrait être appelée pour traiter la requête. Cela vous permet d'une part de spécifier de belles URLs et d'autre part de conserver la fonctionnalité de votre application découplée de ces URLs. Le routage est un mécanisme bidirectionnel, ce qui signifie qu'il doit aussi être utilisé pour générer des URLs.

En savoir plus grâce au Cookbook

- *Comment forcer les routes à toujours utiliser HTTPS ou HTTP*

Chapter 7
Créer et utiliser les templates

Comme vous le savez, le *contrôleur* est responsable de la gestion de toutes les requêtes d'une application Symfony2. En réalité, le contrôleur délègue le plus gros du travail à d'autres classes afin que le code puisse être testé et réutilisé. Quand un contrôleur a besoin de générer du HTML, CSS ou tout autre contenu, il donne la main au moteur de template. Dans ce chapitre, vous apprendrez comment écrire des templates puissants qui peuvent être utilisés pour retourner du contenu à l'utilisateur, remplir le corps d'emails et bien d'autres. Vous apprendrez des raccourcis, des méthodes ingénieuses pour étendre les templates et surtout comment les réutiliser.

L'affichage des templates est abordé dans la page *contrôleur* du Book.

Templates

Un template est un simple fichier texte qui peut générer n'importe quel format basé sur du texte (HTML, XML, CSV, LaTeX ...). Le type de template le plus connu est le template *PHP* - un fichier texte interprété par PHP qui contient du texte et du code PHP:

Listing 7-1

```
<!DOCTYPE html>
<html>
    <head>
        <title>Bienvenue sur Symfony !</title>
    </head>
    <body>
        <h1><?php echo $page_title ?></h1>

        <ul id="navigation">
            <?php foreach ($navigation as $item): ?>
                <li>
                    <a href="<?php echo $item->getHref() ?>">
                        <?php echo $item->getCaption() ?>
                    </a>
```

```
15                 </li>
16             <?php endforeach; ?>
17         </ul>
18     </body>
19 </html>
```

Mais Symfony2 contient un langage de template encore plus puissant appelé *Twig*[1]. Twig vous permet d'écrire des templates simples et lisibles qui sont plus compréhensibles par les web designers et, dans bien des aspects, plus puissants que les templates PHP :

Listing 7-2
```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Bienvenue sur Symfony !</title>
5      </head>
6      <body>
7          <h1>{{ page_title }}</h1>
8
9          <ul id="navigation">
10             {% for item in navigation %}
11                 <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
12             {% endfor %}
13         </ul>
14     </body>
15 </html>
```

Twig définit deux types de syntaxe spéciale :

- `{{ ... }}`: « Dit quelque chose »: écrit une variable ou le résultat d'une expression dans le template ;
- `{% ... %}`: « Fait quelque chose »: un **tag** qui contrôle la logique du template ; il est utilisé pour exécuter des instructions comme la boucle for par exemple.

Il y a une troisième syntaxe utilisée pour les commentaires : `{# un commentaire #}`. Cette syntaxe peut être utilisée sur plusieurs lignes comme la syntaxe PHP `/* commentaire */` qui est équivalente.

Twig contient également des **filtres**, qui modifient le contenu avant de le rendre. Le filtre suivant met la variable `title` en majuscule avant de la rendre :

Listing 7-3
```
1 {{ title | upper }}
```

Twig est fourni avec une longue liste de *tags*[2] et de *filtres*[3] qui sont disponibles par défaut. Vous pouvez même *ajouter vos propres extensions*[4] à Twig si besoin.

Créer une nouvelle extension Twig est aussi simple que de créer un nouveau service et de le taguer avec `twig.extension` *tag*.

1. http://twig.sensiolabs.org
2. http://twig.sensiolabs.org/doc/tags/index.html
3. http://twig.sensiolabs.org/doc/filters/index.html
4. http://twig.sensiolabs.org/doc/advanced.html#creating-an-extension

Comme vous le verrez tout au long de la documentation, Twig supporte aussi les fonctions, et de nouvelles fonctions peuvent être ajoutées. Par exemple, la fonction suivante utilise le tag standard `for` et la fonction `cycle` pour écrire dix balises div en alternant les classes `odd` et `even` :

Listing 7-4
```
{% for i in 0..10 %}
  <div class="{{ cycle(['odd', 'even'], i) }}">
    <!-- some HTML here -->
  </div>
{% endfor %}
```

Tout au long de ce chapitre, les exemples de templates seront donnés à la fois avec Twig et PHP.

Si vous choisissez de ne *pas* utiliser Twig et que vous le désactivez, vous devrez implémenter votre propre gestionnaire d'exceptions via l'évènement `kernel.exception`.

Pourquoi Twig?

Les templates Twig sont conçus pour être simples et ne traiteront aucun code PHP. De par sa conception, le système de template Twig s'occupe de la présentation, pas de la logique. Plus vous utiliserez Twig, plus vous apprécierez cette distinction et en bénéficierez. Et bien sûr, vous serez adoré par tous les web designers.

Twig peut aussi faire des choses que PHP ne pourrait pas faire, comme le contrôle d'espaces blancs, le bac à sable, l'échappement de caractères automatique et contextuel et l'inclusion de fonctions et de filtres personnalisés qui n'affectent que les templates. Twig contient de petites fonctionnalités qui rendent l'écriture de template plus facile et plus concise. Prenez l'exemple suivant, il combine une boucle avec l'instruction logique `if` :

Listing 7-5
```
<ul>
    {% for user in users if user.active %}
        <li>{{ user.username }}</li>
    {% else %}
        <li>Aucun utilisateur trouvé.</li>
    {% endfor %}
</ul>
```

Twig et la mise en cache

Twig est rapide. Chaque template Twig est compilé en une classe PHP natif qui est rendue à l'exécution. Les classes compilées sont stockées dans le répertoire `app/cache/{environment}/twig` (où `{environment}` est l'environnement, par exemple `dev` ou `prod`) et elles peuvent être utiles dans certains cas pour débugguer. Lisez le chapitre *Environnements* pour plus d'informations sur les environnements.

Lorsque le mode `debug` est activé (par exemple en environnement de `dev`), un template Twig sera automatiquement recompilé à chaque fois qu'un changement y sera apporté. Cela signifie que durant le développement, vous pouvez effectuer des modifications dans un template Twig et voir instantanément les changements sans vous soucier de vider le cache.

Lorsque le mode `debug` est désactivé (par exemple en environnement de `prod`), en revanche, vous devrez vider le répertoire de cache Twig afin que le template soit régénéré. Souvenez-vous bien de cela lorsque vous déploirez votre application.

L'héritage de template et layouts

Bien souvent, les templates d'un projet partagent des éléments communs, comme les entêtes, pieds de page et menus latéraux. Dans Symfony2, nous abordons ce problème différemment : un template peut être décoré par un autre. Cela fonctionne exactement comme les classes PHP : l'héritage de template vous permet de bâtir un template « layout » de base qui contient tous les éléments communs de votre site et de définir des **blocs** (comprenez « classe PHP avec des méthodes de base »). Un template enfant peut étendre le template layout et surcharger n'importe lequel de ses blocs (comprenez « une sous-classe PHP qui surcharge certaines méthodes de sa classe parente »).

Tout d'abord, construisez un fichier layout :

Listing 7-6

```twig
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block title %}Test Application{% endblock %}</title>
    </head>
    <body>
        <div id="sidebar">
            {% block sidebar %}
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
            {% endblock %}
        </div>

        <div id="content">
            {% block body %}{% endblock %}
        </div>
    </body>
</html>
```

Bien que les explications sur l'héritage de template concernent Twig, la philosophie est la même pour les templates PHP.

Ce template définit le squelette HTML de base d'un document constitué simplement de deux colonnes. Dans cette exemple, trois espaces `{% block %}` sont définis (`title`, `sidebar` et `body`). Chacun de ces blocs peut être soit surchargé dans un template enfant ou soit conserver leur code d'origine. Ce template peut aussi être rendu directement. Dans ce cas, les blocs `title`, `sidebar` et `body` conserveront simplement les valeurs par défaut utilisées dans ce template.

Un template enfant peut ressembler à cela :

Listing 7-7

```twig
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends '::base.html.twig' %}

{% block title %}My cool blog posts{% endblock %}

{% block body %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
```

```
10      {% endfor %}
11  {% endblock %}
```

Le template parent est identifié grâce à une chaîne de caractères particulière (`::base.html.twig`) qui indique que ce template se trouve dans le dossier `app/Resources/views` du projet. Cette convention de nommage est complètement expliquée dans *Nommage de template et Emplacements*.

La clé de l'héritage de template est la balise `{% extends %}`. Elle indique au moteur de template d'évaluer d'abord le template de base, qui configure le layout et définit plusieurs blocs. Le template enfant est ensuite rendu. Durant ce traitement les blocs parents `title` et `body` sont remplacés par ceux de l'enfant. Dépendant de la valeur de `blog_entries`, la sortie peut ressembler à ceci :

Listing 7-8

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5          <title>Mes billets de blog cools</title>
6      </head>
7      <body>
8          <div id="sidebar">
9              <ul>
10                 <li><a href="/">Accueil</a></li>
11                 <li><a href="/blog">Blog</a></li>
12             </ul>
13         </div>
14
15         <div id="content">
16             <h2>Mon premier post</h2>
17             <p>Le corps du premier post.</p>
18
19             <h2>Un autre post</h2>
20             <p>Le corps du deuxième post.</p>
21         </div>
22     </body>
23 </html>
```

Remarquons que comme le template enfant n'a pas défini le bloc `sidebar`, la valeur du template parent est utilisée à la place. Le contenu d'une balise `{% block %}` d'un template parent est toujours utilisé par défaut.

Vous pouvez utiliser autant de niveaux d'héritage que vous souhaitez. Dans la section suivante, un modèle commun d'héritage à trois niveaux sera expliqué, ainsi que l'organisation des templates au sein d'un projet Symfony2.

Quand on travaille avec l'héritage de templates, il est important de garder ces astuces à l'esprit :

- Si vous utilisez `{% extends %}` dans un template, alors ce doit être la première balise de ce template.
- Plus vous utilisez les balises `{% block %}` dans les templates, mieux c'est. Souvenez-vous, les templates enfants ne doivent pas obligatoirement définir tous les blocs parents, donc créez autant de blocs que vous désirez dans le template de base et attribuez leurs une configuration par défaut. Plus vous avez de blocs dans le template de base, plus le layout sera flexible.
- Si vous vous retrouvez à dupliquer du contenu dans plusieurs templates, cela veut probablement dire que vous devriez déplacer ce contenu dans un `{% block %}` d'un template

parent. Dans certain cas, la meilleure solution peut être de déplacer le contenu dans un nouveau template et de l'`include` (voir *L'inclusion de Templates*).

- Si vous avez besoin de récupérer le contenu d'un bloc d'un template parent, vous pouvez utiliser la fonction `{{ parent() }}`. C'est utile si on souhaite compléter le contenu du bloc parent au lieu de le réécrire totalement :

Listing 7-9
```
{% block sidebar %}
    <h3>Table of Contents</h3>
    ...
    {{ parent() }}
{% endblock %}
```

Nommage de template et Emplacements

Par défaut, les templates peuvent se trouver dans deux emplacements différents :

- `app/Resources/views/` : Le dossier `views` de l'application peut aussi bien contenir le template de base de l'application (c-a-d le layout de l'application) ou les templates qui surchargent les templates des bundles (voir *La Surcharge de templates de Bundle*);
- `path/to/bundle/Resources/views/` : Chaque bundle place leurs templates dans leur dossier `Resources/views` (et sous dossiers). La plupart des templates résident au sein d'un bundle.

Symfony2 utilise une chaîne de caractères au format **bundle**:**controller**:**template** pour les templates. Cela permet plusieurs types de templates, chacun se situant à un endroit spécifique :

- `AcmeBlogBundle:Blog:index.html.twig`: Cette syntaxe est utilisée pour spécifier un template pour une page donnée. Les trois parties de la chaîne de caractères, séparées par deux-points (`:`), signifient ceci :

    - `AcmeBlogBundle`: (*bundle*) le template se trouve dans le `AcmeBlogBundle` (`src/Acme/BlogBundle` par exemple);
    - `Blog`: (*controller*) indique que le template se trouve dans le sous-répertoire `Blog` de `Resources/views`;
    - `index.html.twig`: (*template*) le nom réel du fichier est `index.html.twig`.

    En supposant que le `AcmeBlogBundle` se trouve à `src/Acme/BlogBundle`, le chemin final du layout serait `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`.

- `AcmeBlogBundle::layout.html.twig`: Cette syntaxe fait référence à un template de base qui est spécifique au `AcmeBlogBundle`. Puisque la partie du milieu, « controller », est absente (`Blog` par exemple), le template se trouve à `Resources/views/layout.html.twig` dans `AcmeBlogBundle`.
- `::base.html.twig`: Cette syntaxe fait référence à un template de base d'une application ou layout. Remarquez que la chaîne de caractères commence par deux deux-points (`::`), ce qui signifie que les deux parties *bundle* et *controller* sont absentes. Ce qui signifie que le template ne se trouve dans aucun bundle, mais directement dans le répertoire racine `app/Resources/views/`.

Dans la section *La Surcharge de templates de Bundle*, vous verrez comment les templates interagissent avec `AcmeBlogBundle`. Par exemple, il est possible de surcharger un template en plaçant un template du même nom dans le répertoire `app/Resources/AcmeBlogBundle/views/`. Cela offre la possibilité de surcharger les templates fournis par n'importe quel vendor bundle.

La syntaxe de nommage des templates doit vous paraître familière - c'est la même convention de nommage qui est utilisée pour faire référence à *Pattern de Nommage du Contrôleur*.

Les Suffixes de Template

Le format **bundle**:**controller**:**template** de chaque template spécifie *où* se situe le fichier template. Chaque nom de template a aussi deux extensions qui spécifient le *format* et le *moteur* (engine) pour le template.

- **AcmeBlogBundle:Blog:index.html.twig** - format HTML, moteur de template Twig
- **AcmeBlogBundle:Blog:index.html.php** - format HTML, moteur de template PHP
- **AcmeBlogBundle:Blog:index.css.twig** - format CSS, moteur de template Twig

Par défaut, tout template de Symfony2 peut être écrit soit en Twig ou en PHP, et la dernière partie de l'extension (`.twig` ou `.php` par exemple) spécifie lequel de ces deux *moteurs* sera utilisé. La première partie de l'extension (`.html`, `.css` par exemple) désigne le format final du template qui sera généré. Contrairement au moteur, qui détermine comment Symfony2 parsera le template, il s'agit là simplement une méthode organisationnelle qui est utilisée dans le cas où la même ressource a besoin d'être rendue en HTML (`index.html.twig`), en XML (`index.xml.twig`), ou tout autre format. Pour plus d'informations, lisez la section *Debugguer*.

Les *moteurs* disponibles peuvent être configurés et d'autres moteurs peuvent être ajoutés. Voir *Templating Configuration* pour plus de détails.

Balises et Helpers

Vous avez maintenant compris les bases des templates, comment ils sont nommés et comment utiliser l'héritage de templates. Les parties les plus difficiles sont d'ores et déjà derrière vous. Dans cette section, vous apprendrez à utiliser un ensemble d'outils disponibles pour aider à réaliser les tâches les plus communes avec les templates comme l'inclusion de templates, faire des liens entre des pages et l'inclusion d'images.

Symfony2 regroupe plusieurs paquets dont plusieurs spécialisés dans les balises et fonctions Twig qui facilitent le travail du web designer. En PHP, le système de templates fournit un système de *helper* extensible. Ce système fournit des propriétés utiles dans le contexte des templates.

Nous avons déjà vu quelques balises Twig (`{% block %}` & `{% extends %}`) ainsi qu'un exemple de helper PHP (`$view['slots']`). Apprenons-en un peu plus.

L'inclusion de Templates

Vous voudrez souvent inclure le même template ou fragment de code dans différentes pages. Par exemple, dans une application avec un espace « nouveaux articles », le code du template affichant un article peut être utilisé sur la page détaillant l'article, sur une page affichant les articles les plus populaires, ou dans une liste des derniers articles.

Quand vous avez besoin de réutiliser une grande partie d'un code PHP, typiquement vous déplacez le code dans une nouvelle classe PHP ou dans une fonction. La même chose s'applique aussi aux templates. En déplaçant le code réutilisé dans son propre template, il peut être inclus par tous les autres templates. D'abord, créez le template que vous souhaiterez réutiliser.

Listing 7-10

```twig
{# src/Acme/ArticleBundle/Resources/views/Article/articleDetails.html.twig #}
<h2>{{ article.title }}</h2>
<h3 class="byline">by {{ article.authorName }}</h3>

<p>
  {{ article.body }}
</p>
```

L'inclusion de ce template dans tout autre template est simple :

Listing 7-11
```twig
{# src/Acme/ArticleBundle/Resources/views/Article/list.html.twig #}
{% extends 'AcmeArticleBundle::layout.html.twig' %}

{% block body %}
    <h1>Recent Articles<h1>

    {% for article in articles %}
        {% include 'AcmeArticleBundle:Article:articleDetails.html.twig' with {'article':
article} %}
    {% endfor %}
{% endblock %}
```

Le template est inclu via l'utilisation de la balise `{% include %}`. Remarquons que le nom du template suit la même convention habituelle. Le template `articleDetails.html.twig` utilise une variable `article`. Elle est passée au template `list.html.twig` en utilisant la commande `with`.

La syntaxe `{'article': article}` est la syntaxe standard de Twig pour les tables de hachage (hash maps) (c-a-d un tableau clé-valeurs). Si vous souhaitez passer plusieurs elements, cela ressemblera à ceci : `{'foo': foo, 'bar': bar}`.

Contrôleurs imbriqués

Dans certains cas, vous aurez besoin d'inclure plus qu'un simple template. Supposons que vous avez un menu latéral dans votre layout qui contient les trois articles les plus récents. La récupération des trois articles les plus récents peut nécessiter l'inclusion d'une requête vers une base de données et de réaliser d'autres opérations logiques qui ne peuvent pas être effectuées dans un template.

La solution consiste simplement à imbriquer les résultats d'un contrôleur dans un template. Dans un premier temps, créez un contrôleur qui retourne un certain nombre d'articles récents:

Listing 7-12
```twig
// src/Acme/ArticleBundle/Controller/ArticleController.php

class ArticleController extends Controller
{
    public function recentArticlesAction($max = 3)
    {
        // un appel en base de données ou n'importe quoi qui retourne les "$max" plus
récents articles
        $articles = ...;

        return $this->render('AcmeArticleBundle:Article:recentList.html.twig',
array('articles' => $articles));
    }
}
```

Le template `recentList` est simplement le suivant :

Listing 7-13
```twig
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{% for article in articles %}
  <a href="/article/{{ article.slug }}">
      {{ article.title }}
  </a>
{% endfor %}
```

Notez que dans l'exemple de cet article, les URLs sont codées en dur (`/article/*slug*` par exemple). Ce n'est pas une bonne pratique. Dans la section suivante, vous apprendrez comment le faire correctement.

Pour inclure le contrôleur, vous avez besoin de faire référence à ce dernier en utilisant la chaîne de caractères standard pour les contrôleurs (c-a-d **bundle**:**controller**:**action**) :

Listing 7-14
```twig
{# app/Resources/views/base.html.twig #}
...

<div id="sidebar">
    {{ render(controller('AcmeArticleBundle:Article:recentArticles', { 'max': 3 })) }}
</div>
```

A chaque fois que vous pensez avoir besoin d'une variable ou de quelques informations auxquelles vous n'avez pas accès à partir d'un template, penser à rendre un contrôleur. Les contrôleurs sont rapides à l'exécution et favorisent une bonne organisation et réutilisabilité du code.

Contenu asynchrone avec hinclude.js

New in version 2.1: hinclude.js support was added in Symfony 2.1

Les contrôleurs peuvent être imbriqués de façon asynchrone avec la bibliothèque javascript *hinclude.js*[5]. Comme le contenu imbriqué vient d'une autre page (un d'un autre contrôleur), Symfony2 utiliser le helper standard `render` pour configurer les tags `hinclude`:

Listing 7-15
```twig
{% render '...:news' with {}, {'standalone': 'js'} %}
```

hinclude.js[6] doit être inclus dans votre page pour fonctionner.

Le contenu par défaut (pendant le chargement ou si javascript n'est pas activé) peut être défini de manière globale dans la configuration de votre application :

Listing 7-16
```yaml
# app/config/config.yml
framework:
```

5. http://mnot.github.com/hinclude/
6. http://mnot.github.com/hinclude/

```
    # ...
    templating:
        hinclude_default_template: AcmeDemoBundle::hinclude.html.twig
```

Liens vers des Pages

La création de liens vers d'autres pages de votre projet est l'opération la plus commune qui soit dans un template. Au lieu de coder en dur les URLs dans les templates, utilisez la fonction `path` de Twig (ou le helper `router` en PHP) pour générer les URLs basées sur la configuration des routes. Plus tard, si vous désirez modifier l'URL d'une page particulière, tout ce que vous avez besoin de faire c'est changer la configuration des routes; les templates génèreront automatiquement la nouvelle URL.

Dans un premier temps, configurons le lien vers la page « _welcome » qui est accessible via la configuration de route suivante :

Listing 7-17
```
_welcome:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Welcome:index }
```

Pour faire un lien vers cette page, utilisons simplement la fonction `path` de Twig en faisant référence à cette route :

Listing 7-18
```
<a href="{{ path('_welcome') }}">Home</a>
```

Comme prévu, ceci génèrera l'URL `/`. Voyons comment cela fonctionne avec des routes plus compliquées :

Listing 7-19
```
article_show:
    pattern:  /article/{slug}
    defaults: { _controller: AcmeArticleBundle:Article:show }
```

Dans ce cas, vous devrez spécifier le nom de route (`article_show`) et une valeur pour le paramètre `{slug}`. En utilisant cette route, revoyons le template `recentList` de la section précédente, et faisons les liens vers les articles correctement :

Listing 7-20
```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{% for article in articles %}
  <a href="{{ path('article_show', { 'slug': article.slug }) }}">
      {{ article.title }}
  </a>
{% endfor %}
```

Vous pouvez aussi générer l'URL absolue en utilisant la fonction `url` de Twig :

Listing 7-21
```
<a href="{{ url('_welcome') }}">Home</a>
```

La même chose peut être réalisée dans les templates en PHP en passant un troisième argument à la méthode `generate()` :

Listing 7-22
```
<a href="<?php echo $view['router']->generate('_welcome', array(), true) ?>">Home</a>
```

Liens vers des Fichiers

Les templates font aussi très souvent référence à des images, du Javascript, des feuilles de style et d'autres fichiers. Bien sûr vous pouvez coder en dur le chemin vers ces fichiers (`/images/logo.png` par exemple), mais Symfony2 fournit une façon de faire plus souple via la fonction `asset` de Twig :

Listing 7-23
```
<img src="{{ asset('images/logo.png') }}" alt="Symfony!" />

<link href="{{ asset('css/blog.css') }}" rel="stylesheet" type="text/css" />
```

Le principal objectif de la fonction `asset` est de rendre votre application plus portable. Si votre application se trouve à la racine de votre hôte (*http://example.com*[7] par exemple), alors les chemins se retourné sera `/images/logo.png`. Mais si votre application se trouve dans un sous répertoire (*http://example.com/my_app*[8] par exemple), chaque chemin vers les fichiers sera alors généré avec le sous répertoire (`/my_app/images/logo.png` par exemple). La fonction `asset` fait attention à cela en déterminant comment votre application est utilisée et en générant les chemins corrects.

De plus, si vous utilisez la fonction `asset`, Symfony peut automatiquement ajouter une chaîne de caractères afin de garantir que la ressource statique mise à jour ne sera pas mise en cache lors de son déploiement. Par exemple, `/images/logo.png` pourrait ressembler à `/images/logo.png?v2`. Pour plus d'informations, lisez la documentation de l'option de configuration *assets_version*.

L'inclusion de Feuilles de style et de Javascripts avec Twig

Aucun site n'est complet sans inclure des fichiers Javascript et des feuilles de styles. Dans Symfony, l'inclusion de ces fichiers est gérée d'une façon élégante en conservant les avantages du mécanisme d'héritage de templates de Symfony.

Cette section vous apprendra la philosophie qui existe derrière l'inclusion de feuilles de style et de fichiers Javascript dans Symfony. Symfony contient aussi une autre bibliothèque, appelée Assetic, qui suit la même philosophie, mais vous permet de faire des choses plus intéressantes avec ces fichiers. Pour plus d'informations sur le sujet voir *Comment utiliser Assetic pour gérer vos ressources*.

Commençons par ajouter deux blocs à notre template de base qui incluront deux fichiers : l'un s'appelle `stylesheet` et est inclus dans la balise `head`, et l'autre s'appelle `javascript` et est inclus juste avant que la base `body` ne se referme. Ces blocs contiendront toutes les feuilles de style et tous les fichiers javascript dont vous aurez besoin pour votre site :

Listing 7-24
```
{# 'app/Resources/views/base.html.twig' #}
<html>
    <head>
        {# ... #}

        {% block stylesheets %}
            <link href="{{ asset('/css/main.css') }}" type="text/css" rel="stylesheet" />
        {% endblock %}
    </head>
    <body>
        {# ... #}

```

7. http://example.com
8. http://example.com/my_app

```
        {% block javascripts %}
            <script src="{{ asset('/js/main.js') }}" type="text/javascript"></script>
        {% endblock %}
    </body>
</html>
```

C'est assez simple. Mais comment faire si vous avez besoin d'inclure une feuille de style supplémentaire ou un autre fichier javascript à partir d'un template enfant ? Par exemple, supposons que vous avez une page contact et que vous avez besoin d'inclure une feuille de style `contact.css` uniquement sur cette page. Au sein du template de la page contact, faites comme ce qui suit :

Listing 7-25
```
{# src/Acme/DemoBundle/Resources/views/Contact/contact.html.twig #}
{% extends '::base.html.twig' %}

{% block stylesheets %}
    {{ parent() }}

    <link href="{{ asset('/css/contact.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}

{# ... #}
```

Dans le template enfant, nous surchargeons simplement le bloc `stylesheets` en ajoutant une nouvelle balise de feuille de style dans ce bloc. Bien sûr, puisque nous voulons ajouter du contenu au bloc parent (et non le *remplacer*), nous devons utiliser la fonction Twig `parent()` pour inclure le bloc `stylesheets` du template de base.

Vous pouvez aussi inclure des ressources situées dans le dossier `Resources/public` de vos bundles. Vous devrez lancer la commande `php app/console assets:install target [--symlink]` pour placer les fichiers dans le bon répertoire ("web" par défaut)

Listing 7-26
```
<link href="{{ asset('bundles/acmedemo/css/contact.css') }}" type="text/css"
rel="stylesheet" />
```

Le résultat final est une page qui inclut à la fois la feuille de style `main.css` et `contact.css`.

Configuration et Utilisation du Service `templating`

Le coeur du système de template dans Symfony2 est le `moteur` de template (`Engine`). Cet objet spécial est responsable de rendre des templates et de retourner leur contenu. Quand vous effectuez le rendu d'un template à travers un contrôleur par exemple, vous utilisez en effet le service de moteur de template. Par exemple:

Listing 7-27
```
return $this->render('AcmeArticleBundle:Article:index.html.twig');
```

est équivalent à:

Listing 7-28
```
$engine = $this->container->get('templating');
$content = $engine->render('AcmeArticleBundle:Article:index.html.twig');

return $response = new Response($content);
```

Le moteur de template (ou « service ») est préconfiguré pour fonctionner automatiquement dans Symfony2. Il peut, bien sûr, être configuré grâce au fichier de configuration de l'application :

Listing 7-29
```yaml
# app/config/config.yml
framework:
    # ...
    templating: { engines: ['twig'] }
```

Plusieurs options de configuration sont disponibles et sont détaillées dans le *Configuration Appendix*.

Le moteur `twig` est nécessaire pour utiliser un webprofiler (de même que beaucoup de bundles tiers).

La Surcharge de templates de Bundle

La communauté Symfony2 est fière de créer et de maintenir des bundles de haute qualité (voir *KnpBundles.com*[9]) concernant un grand nombre de fonctionnalités. Une fois que vous utilisez un tel bundle, vous aimeriez sûrement surcharger et personnaliser un ou plusieurs de ses templates.

Supposons que vous utilisiez un imaginaire `AcmeBlogBundle` open source dans votre projet (dans le répertoire `src/Acme/blogBundle` par exemple). Même si vous êtes très content de ce bundle, vous voudriez probablement surcharger la page « liste » du blog pour la personnaliser et l'adapter spécialement à votre application. En cherchant un peu dans le contrôleur `Blog` du `AcmeBlogBundle`, vous trouvez ceci :

Listing 7-30
```php
public function indexAction()
{
    $blogs = // logique qui récupère les blogs

    $this->render('AcmeBlogBundle:Blog:index.html.twig', array('blogs' => $blogs));
}
```

Quand le `AcmeBlogBundle:Blog:index.html.twig` est rendu, Symfony2 regarde en fait dans deux emplacements pour le template :

1. `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig`
2. `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`

Pour surcharger le template du bundle, il suffit de copier le template `index.html.twig` du bundle vers `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig` (le répertoire `app/Resources/AcmeBlogBundle` n'existe pas, nous vous laissons le soin de le créer). Vous êtes maintenant à même de personnaliser le template.

Si vous ajoutez un template à un nouvel endroit, vous *pourriez* avoir besoin de vider votre cache (`php app/console cache:clear`), même si vous êtes en mode debug.

Cette logique s'applique aussi au template layout. Supposons maintenant que chaque template dans `AcmeBlogBundle` hérite d'un template de base appelé `AcmeBlogBundle::layout.html.twig`. Comme précédemment, Symfony2 regardera dans les deux emplacements suivants :

1. `app/Resources/AcmeBlogBundle/views/layout.html.twig`
2. `src/Acme/BlogBundle/Resources/views/layout.html.twig`

9. `http://knpbundles.com`

Encore une fois, pour surcharger le template, il suffit de le copier du bundle vers `app/Resources/AcmeBlogBundle/views/layout.html.twig`. Vous êtes maintenant libre de personnaliser cette copie comme il vous plaira.

Si vous prenez du recul, vous vous apercevrez que Symfony2 commence toujours par regarder dans le répertoire `app/Resources/{BUNDLE_NAME}/views/` pour un template. Si le template n'existe pas là, il continue sa recherche dans le répertoire `Resources/views` du bundle lui-même. Ce qui signifie que tous les templates d'un bundle peuvent être surchargés à condition de les placer dans le bon sous-répertoire de `app/Resources`.

Vous pouvez également surcharger les templates provenant d'un bundle grâce à l'héritage de bundle. Pour plus d'informations, voir *Comment utiliser l'héritage de bundle pour surcharger certaines parties d'un bundle*.

La Surcharge des Core Templates

Puisque le framework Symfony2 lui-même est juste un bundle, les templates du noyau peuvent être surchargés de la même façon. Par exemple, le bundle noyau `TwigBundle` contient un certain nombre de templates relatifs aux « exceptions » et aux « erreurs » qui peuvent être surchargés en copiant chacun d'eux du répertoire `Ressources/views/Exception` du `TwigBundle` vers, vous l'avez deviné, le répertoire `app/Resources/TwigBundle/views/Exception`.

Trois niveaux d'héritages

Une façon commune d'utiliser l'héritage est d'utiliser l'approche à trois niveaux. Cette méthode fonctionne parfaitement avec trois différents types de templates détaillés :

- Créez un fichier `app/Resources/views/base.html.twig` qui contient le principal layout pour votre application (comme dans l'exemple précédent). En interne, ce template est appelé `::base.html.twig`;
- Créez un template pour chaque « section » de votre site. Par exemple, un `AcmeBlogBundle` devrait avoir un template appelé `AcmeBlogBundle::layout.html.twig` qui contient uniquement les éléments de section spécifiques au blog;

Listing 7-31
```
{# src/Acme/BlogBundle/Resources/views/layout.html.twig #}
{% extends '::base.html.twig' %}

{% block body %}
    <h1>Blog Application</h1>

    {% block content %}{% endblock %}
{% endblock %}
```

- Créez un template individuel pour chaque page et faites en sorte que chacun étende (extend) le template approprié à la section. Par exemple, la page « index » sera appelée à peu de chose près `AcmeBlogBundle:Blog:index.html.twig` et listera les billets du blog.

Listing 7-32
```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends 'AcmeBlogBundle::layout.html.twig' %}

```

```
4  {% block content %}
5      {% for entry in blog_entries %}
6          <h2>{{ entry.title }}</h2>
7          <p>{{ entry.body }}</p>
8      {% endfor %}
9  {% endblock %}
```

Remarquons que ce template étend la section (`AcmeBlogBundle::layout.html.twig`) qui à son tour étend le layout de base de l'application (`::base.html.twig`). C'est le modèle commun d'héritage à trois niveaux.

Quand vous construisez votre application, vous pouvez choisir de suivre cette méthode ou simplement faire que chaque template de page étende le layout de l'application directement (`{% extends '::base.html.twig' %}` par exemple). Le modèle des trois templates est une meilleure pratique, utilisée par les bundles vendor. De ce fait, le layout d'un bundle peut facilement être surchargé pour étendre proprement le layout de base de votre application.

L'Échappement

Lors de la génération HTML d'un template, il y a toujours un risque qu'une variable du template affiche du code HTML non désiré ou du code dangereux côté client. Le résultat est que le contenu dynamique peut casser le code HTML de la page résultante ou permettre à un utilisateur malicieux de réaliser une attaque *Cross Site Scripting*[10] (XSS). Considérons cet exemple classique :

Listing 7-33
```
1  Hello {{ name }}
```

Imaginons que l'utilisateur ait rentré le code suivant comme son nom :

Listing 7-34
```
1  <script>alert('hello!')</script>
```

Sans échappement du rendu, le template résultant provoquera l'affichage d'une boîte d'alert Javascript :

Listing 7-35
```
1  Hello <script>alert('hello!')</script>
```

Et bien que cela semble inoffensif, si un utilisateur peut aller aussi loin, ce même utilisateur peut aussi écrire un Javascript qui réalise des actions malicieuses dans un espace sécurisé d'un inconnu et légitime utilisateur.

La réponse à ce problème est l'échappement (output escaping). En activant l'échappement, le même template sera rendu de façon inoffensive, et affichera littéralement la balise `script` à l'écran :

Listing 7-36
```
1  Hello &lt;script&gt;alert(&#39;helloe&#39;)&lt;/script&gt;
```

Les systèmes de template Twig et PHP abordent le problème différemment. Si vous utilisez Twig, l'échappement est activé par défaut et vous êtes protégé. En PHP, l'échappement n'est pas automatique, ce qui signifie que vous aurez besoin d'échapper manuellement là où c'est nécessaire.

10. `http://en.wikipedia.org/wiki/Cross-site_scripting`

L'échappement avec Twig

Si vous utilisez les templates Twig, alors l'échappement est activé par défaut. Ce qui signifie que vous êtes protégé immédiatement des conséquences non intentionnelles du code soumis par l'utilisateur. Par défaut, l'échappement suppose que le contenu est bien échappé pour un affichage HTML.

Dans certains cas, vous aurez besoin de désactiver l'échappement de la sortie lors du rendu d'une variable qui est sure et qui contient des décorations qui ne doivent pas être échappées. Supposons que des utilisateurs administrateurs sont capables décrire des articles qui contiennent du code HTML. Par défaut, Twig échappera le corps de l'article. Pour le rendre normalement, il suffit d'ajouter le filtre `raw` : `{{ article.body | raw }}`.

Vous pouvez aussi désactiver l'échappement au sein d'un `{% block %}` ou pour un template entier. Pour plus d'informations, voir *Output Escaping*[11] dans la documentation de Twig.

L'échappement en PHP

L'échappement n'est pas automatique lorsque vous utilisez des templates PHP. Ce qui signifie que, à moins que vous ne choisissiez explicitement d'échapper une variable, vous n'êtes pas protégé. Pour utiliser l'échappement, utilisez la méthode spéciale `escape()` de view : :

Listing 7-37
```php
Hello <?php echo $view->escape($name) ?>
```

Par défaut, la méthode `escape()` suppose que la variable est rendue dans un contexte HTML (et donc que la variable est échappée pour être sans danger pour l'HTML). Le second argument vous permet de changer de contexte. Par exemple, pour afficher quelque chose dans une chaîne de caractères JavaScript, utilisez le context `js` :

Listing 7-38
```php
var myMsg = 'Hello <?php echo $view->escape($name, 'js') ?>';
```

Debugguer

New in version 2.0.9: Cette fonctionnalité est disponible depuis Twig `1.5.x`, qui a été fourni pour la première fois avec Symfony 2.0.9.

Lorsque vous utilisez PHP, vous pouvez utiliser `var_dump()` si vous avez besoin de trouver rapidement la valeur d'une variable. C'est utile, par exemple, dans un contrôleur. La même chose peut être faite lorsque vous utilisez Twig en utilisant l'extension de debug. Elle a besoin d'être activée dans la configuration :

Listing 7-39
```yaml
# app/config/config.yml
services:
    acme_hello.twig.extension.debug:
        class:        Twig_Extension_Debug
        tags:
            - { name: 'twig.extension' }
```

Les paramètres de template peuvent être affichés en utilisant la fonction `dump` :

Listing 7-40

11. http://twig.sensiolabs.org/doc/api.html#escaper-extension

```twig
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{{ dump(articles) }}

{% for article in articles %}
    <a href="/article/{{ article.slug }}">
        {{ article.title }}
    </a>
{% endfor %}
```

Les variables ne seront affichées que si l'option `debug` de Twig (dans `config.yml`) est à `true`. Cela signifie que par défaut, les variables seront affichées en environnement de `dev` mais pas en environnement de `prod`.

Vérification de la syntaxe

New in version 2.1: La commande `twig:lint` a été ajoutée dans Symfony 2.1

Vous pouvez vérifier les éventuelles erreurs de syntaxe dans les templates Twig en utilisant la commande `twig:lint` :

Listing 7-41

```
# Vous pouvez vérifier par nom de fichier :
$ php app/console twig:lint src/Acme/ArticleBundle/Resources/views/Article/
recentList.html.twig

# ou par répertoire :
$ php app/console twig:lint src/Acme/ArticleBundle/Resources/views

# ou en utilisant le nom du bundle :
$ php app/console twig:lint @AcmeArticleBundle
```

Les Formats de Template

Les templates sont une façon générique de rendre un contenu dans *n'importe quel* format. Et bien que dans la plupart des cas vous utiliserez les templates pour rendre du contenu HTML, un template peut tout aussi facilement générer du JavaScript, du CSS, du XML ou tout autre format dont vous pouvez rêver.

Par exemple, la même « ressource » est souvent rendue dans plusieurs formats différents. Pour rendre la page index d'un article en XML, incluez simplement le format dans le nom du template :

- *nom du template XML*: `AcmeArticleBundle:Article:index.xml.twig`
- *nom de fichier du template XML*: `index.xml.twig`

En réalité, ce n'est rien de plus qu'une convention de nommage et le template n'est pas rendu différemment en se basant sur ce format.

Dans beaucoup de cas, vous pourriez vouloir autoriser un simple contrôleur à rendre plusieurs formats en se basant sur le « request format ». Pour cette raison, un pattern commun est de procéder comme cela:

Listing 7-42

```php
public function indexAction()
{
    $format = $this->getRequest()->getRequestFormat();

    return $this->render('AcmeBlogBundle:Blog:index.'.$format.'.twig');
}
```

Le `getRequestFormat` sur l'objet `Request` retourne par défaut `html`, mais peut aussi retourner n'importe quel autre format basé sur le format demandé par l'utilisateur. Le format demandé est le plus souvent géré par le système de route, où une route peut être configurée telle que `/contact` définit le format demandé à `html` alors que `/contact.xml` définit le format à `xml`. Pour plus d'informations, voir l'*Exemple avancé du chapitre routing*.

Pour créer des liens qui incluent le paramètre de format, incluez une clé `_format` dans le tableau de paramètres :

Listing 7-43
```twig
<a href="{{ path('article_show', {'id': 123, '_format': 'pdf'}) }}">
    PDF Version
</a>
```

Réflexions Finales

Le moteur de template dans Symfony est un outil puissant qui peut être utilisé chaque fois que vous avez besoin de générer du contenu de représentation en HTML, XML ou tout autre format. Et bien que les templates soient un moyen commun de générer du contenu dans un contrôleur, leur utilisation n'est pas systématique. L'objet `Response` retourné par un contrôleur peut être créé avec ou sans utilisation de template:

Listing 7-44
```php
// création d'un objet Response qui contient le rendu d'un template
$response = $this->render('AcmeArticleBundle:Article:index.html.twig');

// création d'un objet Response qui contient un texte simple
$response = new Response('response content');
```

Le moteur de templates de Symfony est très flexible et deux outils de restitution sont disponibles par défaut : les traditionnels templates *PHP* et les élégants et puissants templates *Twig*. Ils supportent tous les deux une hiérarchie des template et sont fournis avec un ensemble riche de fonctions capables de réaliser la plupart des tâches.

Dans l'ensemble, le système de templates doit être pensé comme étant un outil puissant qui est à votre disposition. Dans certains cas, vous n'aurez pas besoin de rendre un template, et dans Symfony2, c'est tout à fait envisageable.

En savoir plus grâce au Cookbook

- *Comment utiliser PHP plutôt que Twig dans les templates*
- *Comment personnaliser les pages d'erreur*
- *Comment écrire une Extension Twig personnalisée*

Chapter 8
Doctrine et les bases de données

L'une des tâches les plus courantes et difficiles pour toute application consiste à lire et à persister des informations dans une base de données. Heureusement, Symfony intègre *Doctrine*[1], une bibliothèque dont le seul but est de vous fournir des outils puissants afin de vous faciliter la tâche. Dans ce chapitre, vous apprendrez les bases de la philosophie de Doctrine et verrez à quel point il peut être facile de travailler avec une base de données.

Doctrine est totalement découplé de Symfony et son utilisation est optionnelle. Ce chapitre est entièrement consacré à l'ORM Doctrine, dont l'objectif est de mapper vos objets avec une base de données relationnelle (comme *MySQL*, *PostGresSQL* ou *Microsoft SQL*). Si vous préférez utiliser des requêtes SQL brutes, c'est facile, et expliqué dans l'article « *Comment utiliser la couche DBAL de Doctrine* » du cookbook

Vous pouvez aussi persister vos données à l'aide de *MongoDB*[2] en utilisant la bibliothèque ODM de Doctrine. Pour plus d'informations, lisez la documentation « *DoctrineMongoDBBundle* ».

Un simple exemple : un produit

La manière la plus facile de comprendre comment Doctrine fonctionne est de le voir en action. Dans cette section, vous allez configurer votre base de données, créer un objet `Product`, le faire persister dans la base de données et le récupérer.

Coder les exemples en même temps

Si vous souhaitez suivre les exemples au fur et à mesure, créer un `AcmeStoreBundle` à l'aide de la commande :

Listing 8-1
```
1 $ php app/console generate:bundle --namespace=Acme/StoreBundle
```

1. http://www.doctrine-project.org/
2. http://www.mongodb.org/

Configurer la base de données

Avant que vous ne soyez réellement prêt, vous devez configurer les paramètres de connexion à votre base de données. Par convention, ces paramètres sont habituellement placés dans le fichier `app/config/parameters.yml` :

Listing 8-2
```yaml
# app/config/parameters.yml
parameters:
    database_driver:    pdo_mysql
    database_host:      localhost
    database_name:      test_project
    database_user:      root
    database_password:  password

# ...
```

Définir la configuration dans `parameters.yml` est juste une convention. Les paramètres définis dans ce fichier sont référencés dans le fichier de configuration principal au moment de configurer Doctrine :

Listing 8-3
```yaml
# app/config/config.yml
doctrine:
    dbal:
        driver:   "%database_driver%"
        host:     "%database_host%"
        dbname:   "%database_name%"
        user:     "%database_user%"
        password: "%database_password%"
```

En stockant ces paramètres de connexion dans un fichier séparé, vous pouvez facilement garder une version différente de ce fichier sur chaque serveur. Vous pouvez aussi stocker la configuration de la base de données (ou n'importe quelle information sensible) en dehors de votre projet, par exemple dans votre configuration Apache. Pour plus d'informations, consultez l'article *Comment configurer les paramètres externes dans le conteneur de services*.

Maintenant que Doctrine connaît vos paramètres de connexion, vous pouvez lui demander de créer votre base de données :

Listing 8-4
```
$ php app/console doctrine:database:create
```

Configurer la base de données en UTF8

Une erreur que font même les développeurs les plus chevronnés est d'oublier de définir un jeu de caractères (charset) et une collation par défaut sur leurs bases de données. Ils se retrouvent alors avec une collation de type latin qui est la valeur par défaut de la plupart des bases de données. Même s'ils pourraient penser à le faire la toute première fois, ils oublient que tout serait à refaire après avoir lancé des commandes telles que :

Listing 8-5
```
$ php app/console doctrine:database:drop --force
$ php app/console doctrine:database:create
```

Il n'y a aucune manière de configurer ces paramètres par défaut dans Doctrine, puisque Doctrine essaye d'être aussi agnostique que possible en terme de configuration. Un moyen de résoudre ce problème est de configurer les valeurs par défaut au niveau du serveur.

Définir UTF8 par défaut pour MySQL est aussi simple que d'ajouter ces quelques lignes à votre fichier de configuration (typiquement `my.cnf`) :

Listing 8-6
```
[mysqld]
collation-server = utf8_general_ci
character-set-server = utf8
```

Si vous voulez utiliser SQLite comme base de données, vous devrez définir le chemin du fichier qui stockera votre base de données :

Listing 8-7
```
# app/config/config.yml
doctrine:
    dbal:
        driver: pdo_sqlite
        path: "%kernel.root_dir%/sqlite.db"
        charset: UTF8
```

Créer une classe entité

Supposons que vous créiez une application affichant des produits. Sans même pensez à Doctrine ou à votre base de données, vous savez déjà que vous aurez besoin d'un objet `Product` représentant ces derniers. Créez cette classe dans le répertoire `Entity` de votre bundle `AcmeStoreBundle`:

Listing 8-8
```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

class Product
{
    protected $name;

    protected $price;

    protected $description;
}
```

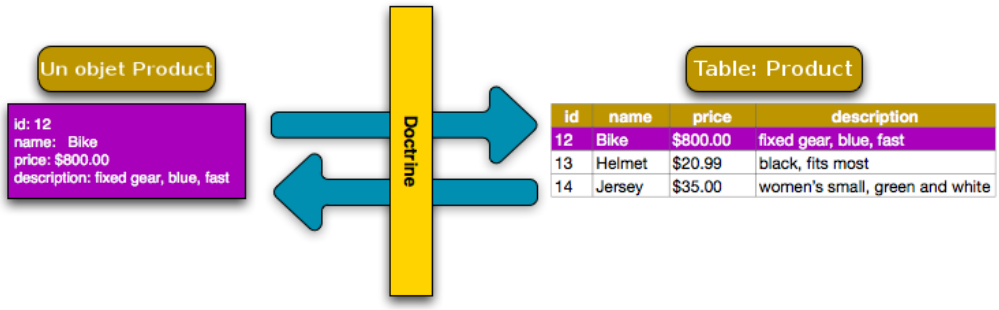
Cette classe - souvent appelée une « entité », ce qui veut dire *une classe basique qui contient des données* - est simple et remplit les besoins métiers des produits dans votre application. Cette classe ne peut pas encore être persistée dans une base de données - c'est juste une simple classe PHP.

Une fois que vous connaissez les concepts derrière Doctrine, vous pouvez l'utiliser pour créer ces classes entité pour vous :

Listing 8-9
```
$ php app/console doctrine:generate:entity \
  --entity="AcmeStoreBundle:Product" \
  --fields="name:string(255) price:float description:text"
```

Ajouter des informations de mapping

Doctrine vous permet de travailler avec des bases de données d'une manière beaucoup plus intéressante que de transformer des lignes en tableaux en vous basant sur des colonnes. Au lieu de ça, Doctrine vous permet de persister des *objets* entiers dans votre base de données et récupérer ces objets depuis votre base de données. Ce système fonctionne en associant vos classes PHP avec des tables de votre base, et les propriétés de ces classes PHP avec des colonnes de la table, c'est ce que l'on appelle le mapping :



Pour que Doctrine soit capable de faire ça, vous n'avez qu'à créer des « métadonnées », ou configurations qui expliquent à Doctrine exactement comment la classe `Product` et ses propriétés doivent être mappées avec la base de données. Ces métadonnées peuvent être spécifiées dans de nombreux formats incluant le YAML, XML ou directement dans la classe `Product` avec les annotations :

Listing 8-10
```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $name;

```

```
24      /**
25       * @ORM\Column(type="decimal", scale=2)
26       */
27      protected $price;
28
29      /**
30       * @ORM\Column(type="text")
31       */
32      protected $description;
33  }
```

Un bundle ne peut accepter qu'un format de définition des métadonnées. Par exemple, il n'est pas possible de mélanger des définitions au format YAML avec des entités annotées dans les classes PHP.

Le nom de la table est optionnel et, s'il est omis, sera déterminé automatiquement en se basant sur le nom de la classe de l'entité.

Doctrine vous permet de choisir parmi une très grande variété de types de champs chacun avec ses propres options. Pour obtenir des informations sur les types de champs disponibles, reportez vous à la section *Référence des types de champs de Doctrine*.

> *Vous pouvez aussi regarder la documentation sur les Bases du Mapping*[3] *de Doctrine pour avoir tout les détails à propos des informations de mapping. Si vous utilisez les annotations, vous devrez préfixer toutes les annotations avec* `ORM\` *(ex:* `ORM\Column(..)`*), ce qui n'est pas montré dans la documentation de Doctrine. Vous devez aussi inclure le morceau de code :* `use Doctrine\ORM\Mapping as ORM;`*, qui importe le préfixe* `ORM` *pour les annotations.*

Faites bien attention que vos noms de classe et de propriétés ne soient pas mappés avec des mots-clés SQL (comme `group` ou `user`). Par exemple, si le nom de la classe de votre entité est `Group` alors, par défaut, le nom de la table correspondante sera `group`, ce qui causera des problèmes SQL avec certains moteurs. Lisez la documentation sur les *Mots-clé SQL réservés*[4] de Doctrine pour savoir comment échapper ces noms. Alternativement, si vous êtes libre de choisir votre schéma de base de données, vous pouvez simplement utiliser un autre nom de table ou de colonne. Lisez les documentations de Doctrine *Classes persistantes*[5] et *Mapping de propriétés*[6].

3. *http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html*
4. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#quoting-reserved-words
5. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/basic-mapping.html#persistent-classes
6. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/basic-mapping.html#property-mapping

Si vous utilisez une autre bibliothèque ou un programme (comme Doxygen) qui utilise les annotations, vous devrez placer une annotation `@IgnoreAnnotation` sur votre classe pour indiquer à Symfony quelles annotations il devra ignorer.

Par exemple, pour empêcher l'annotation `@fn` de lancer une exception, ajouter le code suivant:

Listing 8-11
```
1 /**
2  * @IgnoreAnnotation("fn")
3  */
4 class Product
5 // ...
```

Générer les getters et setters

Même si Doctrine sait maintenant comment persister un objet `Product` dans la base de données, la classe elle-même n'est pas encore très utile. Comme `Product` est juste une simple classe PHP, vous devez créer des getters et des setters (ex: `getName()`, `setName()`) pour pouvoir accéder à ces propriétés (car elles sont `protected`). Heureusement, Doctrine peut faire ça pour vous en lançant la commande :

Listing 8-12
```
1 $ php app/console doctrine:generate:entities Acme/StoreBundle/Entity/Product
```

Cette commande s'assure que tous les getters et les setters sont générés pour la classe `Product`. C'est une commande sure - vous pouvez la lancer encore et encore : elle ne génèrera que les getters et les setters qui n'existent pas (c-à-d qu'elle ne remplace pas les méthodes existantes)

Gardez en tête que le générateur d'entités de Doctrine ne produit que de simples getters/setters. Vous devrez vérifier les entités générées et ajuster la logique des getters/setters selon vos propres besoins.

Un peu plus sur `doctrine:generate:entities`

Avec la commande `doctrine:generate:entities`, vous pouvez :

- générer les getters et setters;
- **générer les classes repository configurées avec les annotations**

  `@ORM\Entity(repositoryClass="...")`;

- générer les constructeurs appropriés pour les relations 1:n et n:m.

La commande `doctrine:generate:entities` fait une sauvegarde de `Product.php` appelée `Product.php~`. Dans certains cas, la présence de ce fichier peut créer l'erreur « Cannot redeclare class ». Vous pouvez supprimer ce fichier en toute sécurité. Vous pouvez aussi utiliser l'option "--no-backup" pour éviter de générer ces fichiers de sauvegarde.

Notez bien que vous n'avez pas *besoin* d'utiliser cette commande. Doctrine ne se base pas sur la génération de code. Comme les classes PHP classiques, vous devez juste vous assurer que vos propriétés protected/private ont bien leurs méthodes getter et setter associées. Comme c'est une tâche récurrente à faire avec Doctrine, cette commande a été créée.

Vous pouvez également générer toutes les entités connues (c-à-d toute classe PHP qui contient des informations de mapping Doctrine) d'un bundle ou d'un namespace :

Listing 8-13

```
$ php app/console doctrine:generate:entities AcmeStoreBundle
$ php app/console doctrine:generate:entities Acme
```

Doctrine se moque que vos propriétés soient `protected` ou `private`, ou même que vous ayez un getter ou un setter pour une propriété. Les getters et setters sont générés ici seulement parce que vous en aurez besoin pour interagir avec vos objets PHP.

Créer les Tables et le Schema

Vous avez maintenant une classe `Product` utilisable avec des informations de mapping permettant à Doctrine de savoir exactement comment le faire persister. Bien sûr, vous n'avez toujours pas la table `product` correspondante dans votre base de données. Heureusement, Doctrine peut créer automatiquement toutes les tables de la base de données nécessaires aux entités connues dans votre application. Pour ce faire, exécutez la commande :

Listing 8-14
```
$ php app/console doctrine:schema:update --force
```

En fait, cette commande est incroyablement puissante. Elle compare ce à quoi votre base de données *devrait* ressembler (en se basant sur le mapping de vos entités) à ce à quoi elle ressemble *vraiment*, et génère le code SQL nécéssaire pour *mettre à jour* la base de données vers ce qu'elle doit être. En d'autres termes, si vous ajoutez une nouvelle propriété avec des métadonnées mappées sur `Product` et relancez cette tâche, elle vous génèrera une requête « alter table » nécessaire pour ajouter cette nouvelle colonne à la table `products` existante.

Une façon encore meilleure de profiter de cette fonctionnalité est d'utiliser les *migrations*, qui vous permettent de générer ces requêtes SQL et de les stocker dans des classes de migration qui peuvent être lancées systématiquement sur vos serveurs de production dans le but de traquer et de migrer vos schémas de base de données de manière sure et fiable.

Votre base de données a maintenant une table `product` totalement fonctionnelle avec des colonnes qui correspondent aux métadonnées que vous avez spécifiées.

Persister des objets dans la base de données

Maintenant que vous avez mappé l'entité `Product` avec la table `product` correspondante, vous êtes prêt à faire persister des données dans la base de données. Depuis un contrôleur, c'est très facile. Ajoutez la méthode suivante au `DefaultController` du bundle :

Listing 8-15
```
// src/Acme/StoreBundle/Controller/DefaultController.php

// ...
use Acme\StoreBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;

public function createAction()
{
    $product = new Product();
    $product->setName('A Foo Bar');
    $product->setPrice('19.99');
    $product->setDescription('Lorem ipsum dolor');

```

```
14      $em = $this->getDoctrine()->getManager();
15      $em->persist($product);
16      $em->flush();
17
18      return new Response('Id du produit créé : '.$product->getId());
19  }
```

Si vous suivez les exemples au fur et à mesure, vous aurez besoin de créer une route qui pointe vers cette action pour voir si elle fonctionne.

Décortiquons cet exemple :

- **lignes 9 à 12** Dans cette section, vous instanciez et travaillez avec l'objet `product` comme n'importe quel autre objet PHP normal.
- **ligne 14** Cette ligne récupère un objet *gestionnaire d'entités* (entity manager) de Doctrine, qui est responsable de la gestion du processus de persistance et de récupération des objets vers et depuis la base de données.
- **ligne 15** La méthode `persist()` dit à Doctrine de « gérer » l'objet `product`. Cela ne crée pas vraiment de requête dans la base de données (du moins pas encore).
- **ligne 16** Quand la méthode `flush()` est appelée, Doctrine regarde dans tous les objets qu'il gère pour savoir si ils ont besoin d'être persistés dans la base de données. Dans cet exemple, l'objet `$product` n'a pas encore été persisté, le gestionnaire d'entités exécute donc une requête `INSERT` et une ligne est créée dans la table `product`.

En fait, comme Doctrine a connaissance de toutes vos entités gérées, lorsque vous appelez la méthode `flush()`, il calcule un ensemble de changements global et exécute la ou les requêtes les plus efficaces possible. Par exemple, si vous persistez un total de 100 objets `Product` et que vous appelez ensuite la méthode `flush()`, Doctrine créera une *unique* requête préparée et la réutilisera pour chaque insertion. Ce concept est nommé *Unité de travail*, et est utilisé pour sa rapidité et son efficacité.

Pour la création et la suppression d'objet, le fonctionnement est le même. Dans la prochaine section, vous découvrirez que Doctrine est assez rusé pour générer une requête `UPDATE` si l'enregistrement est déjà présent dans la base de données.

Doctrine fournit une bibliothèque qui vous permet de charger de manière automatisée des données de test dans votre projet (des « fixtures »). Pour plus d'informations, lisez *DoctrineFixturesBundle*.

Récupérer des objets dans la base de données

Récupérer un objet depuis la base de données est encore plus facile. Par exemple, supposons que vous avez configuré une route pour afficher un `Product` spécifique en se basant sur la valeur de son `id`:

Listing 8-16
```
1  public function showAction($id)
2  {
3      $product = $this->getDoctrine()
4          ->getRepository('AcmeStoreBundle:Product')
5          ->find($id);
6
```

```
    if (!$product) {
        throw $this->createNotFoundException(
            'Aucun produit trouvé pour cet id : '.$id
        );
    }

    // ... faire quelque chose comme envoyer l'objet $product à un template
}
```

Vous pouvez réaliser la même chose sans écrire de code en utilisant le raccourci `@ParamConverter`. Pour plus de détails, lisez la *documentation du FrameworkExtraBundle*.

Lorsque vous requêtez pour un type particulier d'objet, vous utiliserez toujours ce qui est connu sous le nom de « dépôt » (ou « repository »). Dites-vous qu'un dépôt est une classe PHP dont le seul travail est de vous aider à récupérer des entités d'une certaine classe. Vous pouvez accéder au dépôt d'une classe d'entités avec:

Listing 8-17
```
$repository = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product');
```

La chaîne `AcmeStoreBundle:Product` est un raccourci que vous pouvez utiliser n'importe ou dans Doctrine au lieu du nom complet de la classe de l'entité (c.à.d `Acme\StoreBundle\Entity\Product`). Tant que vos entités sont disponibles sous l'espace de nom `Entity` de votre bundle, cela fonctionnera.

Une fois que vous disposez de votre dépôt, vous pouvez accéder à toute sorte de méthodes utiles:

Listing 8-18
```
// requête par clé primaire (souvent "id")
$product = $repository->find($id);

// Noms de méthodes dynamiques en se basant sur un nom de colonne
$product = $repository->findOneById($id);
$product = $repository->findOneByName('foo');

// trouver *tous* les produits
$products = $repository->findAll();

// trouver un groupe de produits en se basant sur une valeur de colonne
$products = $repository->findByPrice(19.99);
```

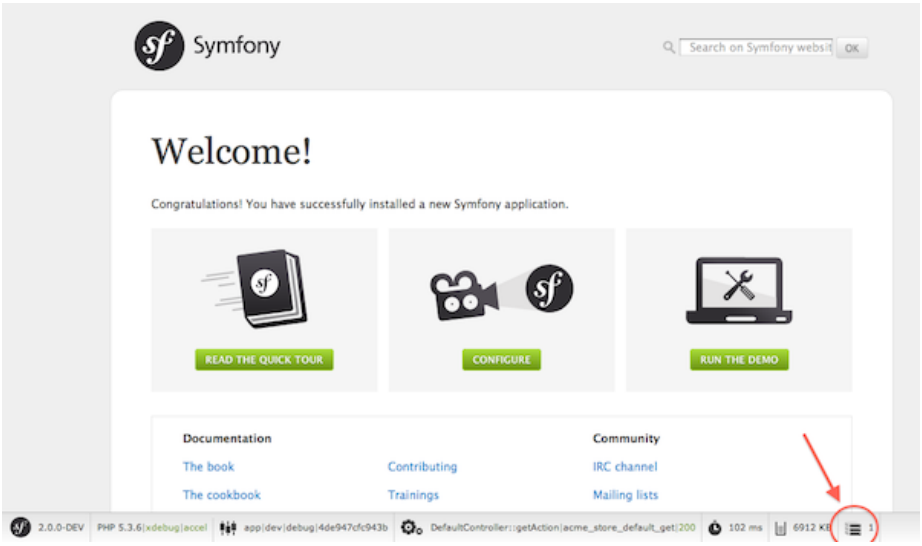
Bien sûr, vous pouvez aussi générer des requêtes complexes, ce que vous apprendrez dans la section *Requêter des objets*.

Vous pouvez aussi profiter des méthodes utiles `findBy` et `findOneBy` pour récupérer facilement des objets en vous basant sur des conditions multiples:

Listing 8-19
```
// requête un seul produit correspondant à un nom et un prix
$product = $repository->findOneBy(array('name' => 'foo', 'price' => 19.99));
```

```
3
4  // requête tout les produits correspondant à un nom, classés par prix
5  $products = $repository->findBy(
6      array('name' => 'foo'),
7      array('price' => 'ASC')
8  );
```

Lorsque vous affichez une page, vous pouvez voir combien de requêtes sont faites dans le coin en bas à droite de votre barre d'outils de débuggage.



Si vous cliquez sur l'icône, le profileur s'ouvrira, vous montrant les requêtes exactes qui ont été faites.

Mettre un objet à jour

Une fois que vous avez récupéré un objet depuis Doctrine, le mettre à jour est facile. Supposons que vous avez une route qui mappe l'id d'un produit vers une action de mise à jour dans un contrôleur:

Listing 8-20

```
1  public function updateAction($id)
2  {
3      $em = $this->getDoctrine()->getManager();
4      $product = $em->getRepository('AcmeStoreBundle:Product')->find($id);
5
6      if (!$product) {
7          throw $this->createNotFoundException(
8              'Aucun produit trouvé pour cet id : '.$id
9          );
10     }
11
12     $product->setName('Nom du nouveau produit!');
13     $em->flush();
14
15     return $this->redirect($this->generateUrl('homepage'));
16 }
```

Mettre à jour l'objet ne nécessite que trois étapes :

1. Récupérer l'objet depuis Doctrine;

2. Modifier l'objet;
3. Apeller la méthode `flush()` du gestionnaire d'entités

Notez qu'appeler `$em->persist($product)` n'est pas nécessaire. Souvenez-vous que cette méthode dit simplement à Doctrine de gérer, ou « regarder » l'objet `$product`. Dans ce cas, comme vous avez récupéré l'objet `$product` depuis Doctrine, il est déjà surveillé.

Supprimer un objet

Supprimer un objet est très similaire, mais requiert un appel à la méthode `remove()` du gestionnaire d'entités :

Listing 8-21
```php
$em->remove($product);
$em->flush();
```

Comme vous vous en doutez, la méthode `remove()` signale à Doctrine que vous voulez supprimer l'entité de la base de données. La vraie requête `DELETE`, cependant, n'est réellement exécutée que lorsque la méthode `flush()` est appelée.

Requêter des objets

Vous avez déjà vu comment les objets dépôts vous permettaient d'exécuter des requêtes basiques sans aucun travail:

Listing 8-22
```php
$repository->find($id);

$repository->findOneByName('Foo');
```

Bien sûr, Doctrine vous permet également d'écrire des requêtes plus complexes en utilisant le Doctrine Query Language (DQL). Le DQL est très ressemblant au SQL excepté que vous devez imaginer que vous requêtez un ou plusieurs objets d'une classe d'entité (ex: `Product`) au lieu de requêter des lignes dans une table (ex: `product`).

Lorsque vous effectuez une requête à l'aide de Doctrine, deux options s'offrent à vous : écrire une requête Doctrine pure ou utilisez le constructeur de requête.

Requêter des objets avec DQL

Imaginons que vous souhaitez récupérer tous les produits dont le prix est supérieur à `19.99`, triés du moins cher au plus cher. Depuis un contrôleur, vous pouvez faire :

Listing 8-23
```php
$em = $this->getDoctrine()->getEntityManager();
$query = $em->createQuery(
    'SELECT p
    FROM AcmeStoreBundle:Product p
    WHERE p.price > :price
    ORDER BY p.price ASC'
)->setParameter('price', '19.99');

$products = $query->getResult();
```

Si vous êtes à l'aise avec SQL, DQL ne devrait pas vous poser de problème. La plus grosse différence est que vous devez penser en terme d'« objets » au lieu de lignes dans une base de données. Pour cette raison, vous effectuez une sélection *depuis* `AcmeStoreBundle:Product` et lui donnez `p` pour alias.

La méthode `getResult()` retourne un tableau de résultats. Si vous ne souhaitez obtenir qu'un seul objet, vous pouvez utiliser la méthode `getSingleResult()` à la place:

Listing 8-24
```
$product = $query->getSingleResult();
```

La méthode `getSingleResult()` lève une exception `Doctrine\ORM\NoResultException` si aucun résultat n'est retourné et une exception `Doctrine\ORM\NonUniqueResultException` si *plus* d'un résultat est retourné. Si vous utilisez cette méthode, vous devrez sans doute l'entourer d'un bloc try/catch pour vous assurer que seul un résultat est retourné (si vous requêtez quelque chose qui pourrait retourner plus d'un résultat):

Listing 8-25
```
$query = $em->createQuery('SELECT ...')
    ->setMaxResults(1);

try {
    $product = $query->getSingleResult();
} catch (\Doctrine\Orm\NoResultException $e) {
    $product = null;
}
// ...
```

La syntaxe du DQL est incroyablement puissante, vous permettant d'effectuer simplement des jointures entre vos entités (le sujet des *relations* sera abordé plus tard), regrouper, etc. Pour plus d'informations, reportez-vous à la documentation officielle de Doctrine : *Doctrine Query Language*[7].

Définir des paramètres

Notez la présence de la méthode `setParameter()`. En travaillant avec Doctrine, la bonne pratique est de définir toutes les valeurs externes en tant que « emplacements », ce qui a été fait dans la requête ci-dessus :

Listing 8-26
```
... WHERE p.price > :price ...
```

Vous pouvez alors définir la valeur de l'emplacement `price` en appelant la méthode `setParameter()`:

Listing 8-27
```
->setParameter('price', '19.99')
```

Utiliser des paramètres au lieu de placer les valeurs directement dans la chaîne constituant la requête permet de se prémunir des attaques de type injections de SQL et devrait *toujours* être fait. Si vous utilisez plusieurs paramètres, vous pouvez alors définir leurs valeurs d'un seul coup en utilisant la méthode `setParameters()`:

Listing 8-28
```
->setParameters(array(
    'price' => '19.99',
    'name'  => 'Foo',
))
```

7. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html

Utiliser le constructeur de requêtes de Doctrine

Au lieu d'écrire des requêtes directement, vous pouvez alternativement utiliser le `QueryBuilder` (constructeur de requêtes) de Doctrine pour faire le même travail en utilisant une jolie interface orientée-objet. Si vous utilisez un IDE, vous pourrez aussi profiter de l'auto-complétion en tapant le nom des méthodes. De l'intérieur d'un contrôleur:

Listing 8-29

```
$repository = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product');

$query = $repository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
    ->orderBy('p.price', 'ASC')
    ->getQuery();

$products = $query->getResult();
```

L'objet `QueryBuilder` contient toutes les méthodes nécessaires pour construire votre requête. En appelant la méthode `getQuery()`, le constructeur de requêtes retourne un objet standard `Query`, qui est identique à celui que vous avez construit dans la section précédente.

Pour plus d'informations sur le constructeur de requêtes de Doctrine, consultez la documentation de Doctrine: *Query Builder*[8]

Classes de dépôt personnalisées

Dans les sections précédentes, vous avez commencé à construire et utiliser des requêtes plus complexes à l'intérieur de vos contrôleurs. Dans le but d'isoler, de tester et de réutiliser ces requêtes, il est conseillé de créer des dépôts personnalisés pour vos entités et d'y ajouter les méthodes contenant vos requêtes.

Pour ce faire, ajouter le nom de la classe dépôt à vos informations de mapping.

Listing 8-30

```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="Acme\StoreBundle\Entity\ProductRepository")
 */
class Product
{
    //...
}
```

Doctrine peut générer la classe de dépôt pour vous en lançant la même commande que celle utilisée précédemment pour générer les getters et setters :

Listing 8-31

```
$ php app/console doctrine:generate:entities Acme
```

Ensuite, ajoutez une méthode `findAllOrderedByName()` à la classe fraîchement générée. Cette méthode requêtera les entités `Product`, en les classant par ordre alphabétique.

Listing 8-32

8. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html

```php
// src/Acme/StoreBundle/Entity/ProductRepository.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery(
                'SELECT p FROM AcmeStoreBundle:Product p ORDER BY p.name ASC'
            )
            ->getResult();
    }
}
```

Vous pouvez accéder au gestionnaire d'entités par `$this->getEntityManager()` à l'intérieur du dépôt.

Vous pouvez alors utiliser cette nouvelle méthode comme les méthodes par défaut du dépôt:

Listing 8-33
```php
$em = $this->getDoctrine()->getManager();
$products = $em->getRepository('AcmeStoreBundle:Product')
            ->findAllOrderedByName();
```

En utilisant un dépôt personnalisé, vous avez toujours accès aux méthodes par défaut telles que `find()` et `findAll()`.

Relations et associations entre les entités

Supposons que les produits de votre application appartiennent tous à exactement une « catégorie ». Dans ce cas, vous aurez besoin d'un objet `Category` et d'une manière de rattacher un objet `Product` à un objet `Category`. Commencez par créer l'entité `Category`. Puisque vous savez que vous aurez besoin que Doctrine persiste votre classe, vous pouvez le laisser générer la classe pour vous.

Listing 8-34
```bash
$ php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Category" \
  --fields="name:string(255)"
```

Cette commande génère l'entité `Category` pour vous, avec un champ `id`, un champ `name` et les méthodes getter et setter associées.

Métadonnées de mapping de relations

Pour relier les entités `Category` et `Product`, commencez par créer une propriété `products` dans la classe `Category` :

Listing 8-35

```php
// src/Acme/StoreBundle/Entity/Category.php

// ...
use Doctrine\Common\Collections\ArrayCollection;

class Category
{
    // ...

    /**
     * @ORM\OneToMany(targetEntity="Product", mappedBy="category")
     */
    protected $products;

    public function __construct()
    {
        $this->products = new ArrayCollection();
    }
}
```

Tout d'abord, comme un objet `Category` sera relié à plusieurs objets `Product`, une propriété `products` (un tableau) est ajoutée pour stocker ces objets `Product`. Encore une fois, nous ne faisons pas cela parce que Doctrine en a besoin, mais plutôt parce qu'il est cohérent dans l'application que chaque `Category` contienne un tableau d'objets `Product`.

Le code de la méthode `__construct()` est important, car Doctrine requiert que la propriété `$products` soit un objet de type `ArrayCollection`. Cet objet ressemble et se comporte *exactement* comme un tableau, mais avec quelque flexibilités supplémentaires. Si ça vous dérange, ne vous inquiétez pas. Imaginez juste que c'est un `array` et vous vous porterez bien.

La valeur targetEntity utilisée plus haut peut faire référence à n'importe quelle entité avec un espace de nom valide, et pas seulement les entités définies dans la même classe. Pour lier une entitée définie dans une autre classe ou un autre bundle, entrez l'espace de nom complet dans targetEntity.

Ensuite, comme chaque classe `Product` est reliée exactement à un objet `Category`, il serait bon d'ajouter une propriété `$category` à la classe `Product` :

Listing 8-36

```php
// src/Acme/StoreBundle/Entity/Product.php

// ...
class Product
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="products")
     * @ORM\JoinColumn(name="category_id", referencedColumnName="id")
     */
    protected $category;
}
```

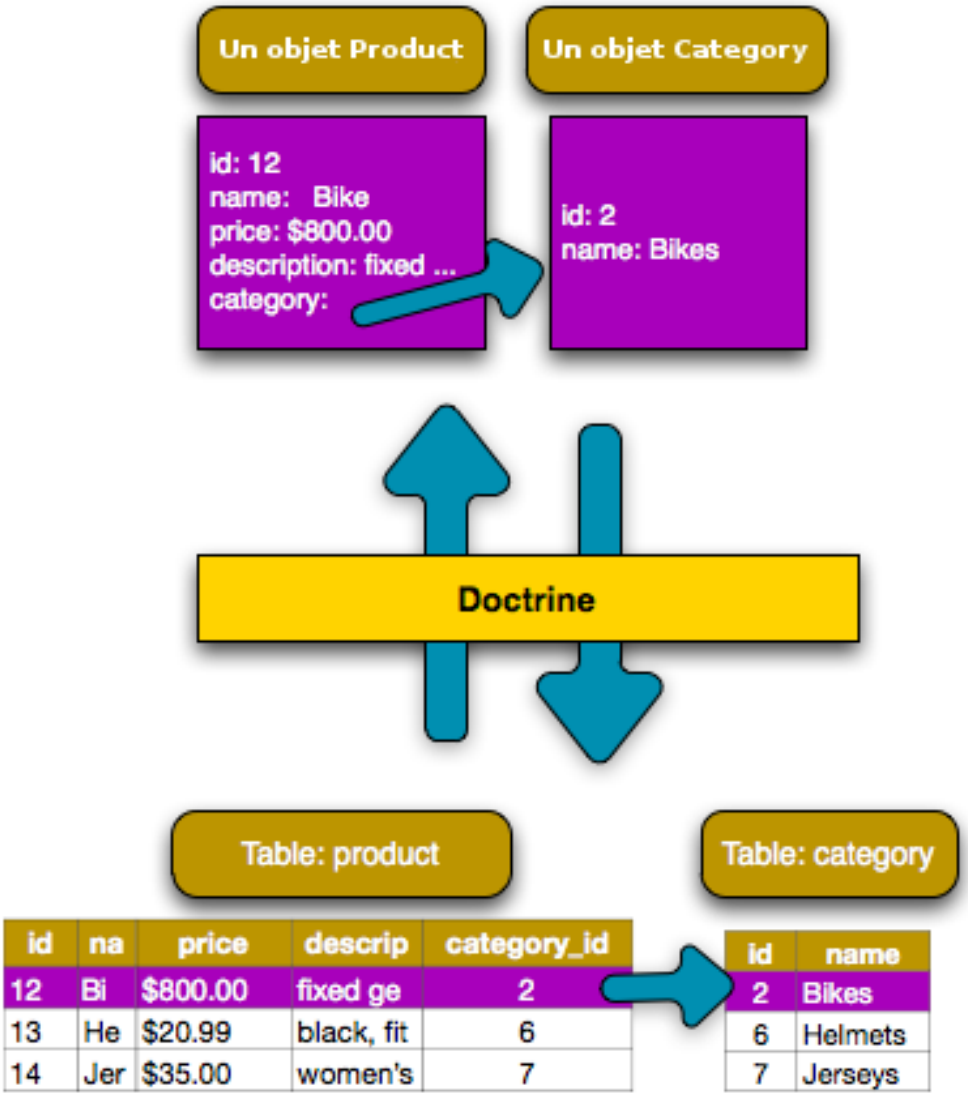
Finalement, maintenant que vous avez ajouté une nouvelle propriété aux classes `Category` et `Product`, dites à Doctrine de régénérer les getters et setters manquants pour vous :

Listing 8-37

```
1 $ php app/console doctrine:generate:entities Acme
```

Laissez de côté les métadonnées de Doctrine pour un moment. Vous avez maintenant deux classes : `Category` et `Product` avec une relation naturelle one-to-many. La classe `Category` peut contenir un tableau de `Product` et l'objet `Product` peut contenir un objet `Category`. En d'autres termes, vous avez construit vos classes de manière à ce qu'elles aient un sens pour répondre à vos besoins. Le fait que les données aient besoin d'être persistées dans une base de données est toujours secondaire.

Maintenant, regardez les métadonnées situées au-dessus de la propriété `$category` dans la classe `Product`. Ces informations indiquent à Doctrine que la classe associée est `Category` et que Doctrine devrait stocker l'`id` de la catégorie dans un champ `category_id` présent dans la table `product`. En d'autres termes, l'objet `Category` associé sera stocké dans la propriété `$category`, mais, de façon transparente, Doctrine persistera la relation en stockant la valeur de l'id de la catégorie dans la colonne `category_id` de la table `product`.



Les métadonnées de la propriété `$products` de l'objet `Category` sont moins importantes, et indiquent simplement à Doctrine de regarder la propriété `Product.category` pour comprendre comment l'association est mappée.

Avant que vous ne continuiez, assurez-vous que Doctrine ajoute la nouvelle table `category`, et la colonne `product.category_id`, ainsi que la nouvelle clé étrangère :

Listing 8-38
```
$ php app/console doctrine:schema:update --force
```

Cette commande ne devrait être exécutée que lors du développement. Pour une façon plus robuste de mettre à jour les bases de données de production, lisez l'article suivant: *Doctrine migrations*.

Sauver les entités associées

Maintenant, pour voir le code en action, imaginez que vous êtes dans un contrôleur :

Listing 8-39
```php
// ...

use Acme\StoreBundle\Entity\Category;
use Acme\StoreBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    public function createProductAction()
    {
        $category = new Category();
        $category->setName('Main Products');

        $product = new Product();
        $product->setName('Foo');
        $product->setPrice(19.99);
        // lie ce produit à une catégorie
        $product->setCategory($category);

        $em = $this->getDoctrine()->getManager();
        $em->persist($category);
        $em->persist($product);
        $em->flush();

        return new Response(
            'Id du produit créé : '.$product->getId().' et id de la catégorie :
'.$category->getId()
        );
    }
}
```

Maintenant, une simple ligne est ajoutée aux tables `category` et `product`. La colonne `product.category_id` du nouveau produit est définie comme la valeur de l'`id` de la nouvelle catégorie. Doctrine gère la persistence de cette relation pour vous.

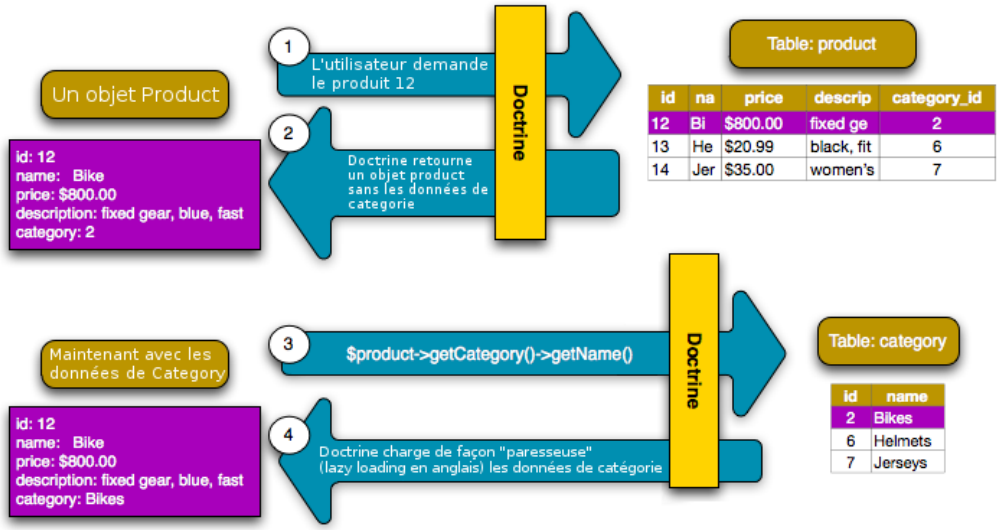
Récupérer des objets associés

Lorsque vous récupérez des objets associés, le processus que vous employez ressemble exactement à celui employé auparavant. Tout d'abord, récupérez un objet `$product` pour accéder à sa `Category` associée:

Listing 8-40

```php
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    $categoryName = $product->getCategory()->getName();

    // ...
}
```

Dans cet exemple, vous requêtez tout d'abord un objet `Product` en vous basant sur l'`id` du produit. Cela genère une requête *uniquement* pour les données du produit et hydrate l'objet `$product` avec ces données. Plus tard, lorsque vous appelez `$product->getCategory()->getName()`, Doctrine effectue une seconde requête de façon transparente pour trouver la `Category` qui est associé à ce `Product`. Il prépare l'objet `$category` et vous le renvoie.



Le plus important est que vous accédiez à la catégorie associée au produit, mais que les données de cette catégorie ne sont réellement récupérées que lorsque vous demandez la catégorie (on parle alors de chargement fainéant ou « lazy loading »).

Vous pouvez aussi faire cette requête dans l'autre sens:

Listing 8-41
```php
public function showProductAction($id)
{
    $category = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Category')
        ->find($id);

    $products = $category->getProducts();

    // ...
}
```

Dans ce cas, la même chose se produit : vous requêtez tout d'abord un simple objet `Category`, et Doctrine effectue alors une seconde requête pour récupérer les objets `Product` associés, mais uniquement une fois

que/si vous les demandez (c-à-d si vous appelez `->getProducts()`). La variable `$products` est un tableau de tous les objets `Product` associés à l'objet `Category` donnés via leurs valeurs `category_id`.

Associations et classes mandataires

Ce mécanisme de « chargement fainéant » est possible car, quand c'est nécessaire, Doctrine retourne un objet « mandataire » (proxy) au lieu des vrais objets. Regardez de plus près l'exemple ci-dessous:

Listing 8-42
```php
$product = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product')
    ->find($id);

$category = $product->getCategory();

// affiche "Proxies\AcmeStoreBundleEntityCategoryProxy"
echo get_class($category);
```

Cet objet mandataire étend le vrai objet `Category`, et à l'air de se comporter exactement de la même manière. La différence est que, en utilisant un objet mandataire, Doctrine peut retarder le requêtage des vraies données de la `Category` jusqu'a ce que vous en ayez réellement besoin (en appelant par exemple `$category->getName()`).

Les classes mandataires sont générées par Doctrine et stockées dans le répertoire du cache. Même si vous ne remarquerez probablement jamais que votre objet `$category` est en fait un objet mandataire, il est important de le garder à l'esprit.

Dans la prochaine section, lorsque vous récupérerez les données du produit et de la catégorie d'un seul coup (via un *join*), Doctrine retournera un *vrai* objet `Category`, car rien ne sera chargé de manière fainéante.

Faire des jointures avec des enregistrements associés

Dans les exemples ci-dessus, deux requêtes ont été faites - une pour l'objet original (par exemple, une `Category`), et une pour le(s) objet(s) associé(s) (par exemple, les objets `Product`)

N'oubliez pas que vous pouvez voir toutes les requêtes effectuées en utilisant la barre d'outils de débuggage.

Bien sûr, si vous savez dès le début que vous aurez besoin d'accéder aux deux objets, vous pouvez éviter de produire une deuxième requête en ajoutant une jointure dans la requête originale. Ajouter le code suivant à la classe `ProductRepository`:

Listing 8-43
```php
// src/Acme/StoreBundle/Entity/ProductRepository.php
public function findOneByIdJoinedToCategory($id)
{
    $query = $this->getEntityManager()
        ->createQuery('
            SELECT p, c FROM AcmeStoreBundle:Product p
            JOIN p.category c
            WHERE p.id = :id'
        )->setParameter('id', $id);

    try {
        return $query->getSingleResult();
```

```
    } catch (\Doctrine\ORM\NoResultException $e) {
        return null;
    }
}
```

Maintenant, vous pouvez utiliser cette méthode dans votre contrôleur pour requêter un objet `Product` et sa `Category` associée avec une seule requête:

Listing 8-44

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->findOneByIdJoinedToCategory($id);

    $category = $product->getCategory();

    // ...
}
```

Plus d'informations sur les associations

Cette section a introduit le type le plus commun d'associations entre les entités, la relation one-to-many. Pour plus de détails et d'exemples avancés sur comment utiliser les autre types de relations (comme `one-to-one`, ou `many-to-many`), consultez la documentation de Doctrine: *Association Mapping Documentation*[9].

Si vous utilisez les annotations, vous devrez préfixer les annotations avec `ORM\` (par exemple: `ORM\OneToMany`), ce qui n'est pas spécifié dans la documentation de Doctrine. Vous aurez aussi besoin d'inclure la ligne `use Doctrine\ORM\Mapping as ORM;` pour *importer* le préfixe d'annotation `ORM`.

Configuration

Doctrine est entièrement configurable, même si vous n'aurez sans doute jamais besoin de vous embêter avec la plupart de ses options. Pour obtenir des informations sur la configuration de Doctrine, rendez-vous dans la section correspondante du *manuel de référence*.

Callbacks et cycle de vie

Parfois, vous voudrez effectuer des actions juste avant ou après qu'une entité ait été insérée, mise à jour ou supprimée. Ces actions sont connues sous le nom de callbacks du « cycle de vie » (lifecycle), car il s'agit de callbacks (méthodes) qui peuvent être appelés à divers moments du cycle de vie de votre entité (par exemple lorsque l'entité est insérée, mise à jour, supprimée, etc.).

Si vous utilisez des annotations pour vos métadonnées, commencez par activer les callbacks du cycle de vie. Si vous utilisez YAML ou XML pour votre mapping, ce n'est pas nécessaire :

Listing 8-45

9. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html

```php
/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Product
{
    // ...
}
```

Désormais, vous pouvez dire à Doctrine d'éxecuter une méthode à n'importe quel évènement du cycle de vie. Par exemple, supposons que vous souhaitez définir une date `created` à la date courante, uniquement lorsque l'entité est persistée pour la première fois (c-à-d insérée) :

Listing 8-46

```php
/**
 * @ORM\PrePersist
 */
public function setCreatedValue()
{
    $this->created = new \DateTime();
}
```

L'exemple ci-dessus suppose que vous avez créé et mappé une propriété `created` (qui n'est pas montrée ici).

Maintenant, juste avant que l'entité soit initialement persistée, Doctrine appelera automatiquement la méthode et le champ `created` sera défini à la date courante.

Vous pouvez procéder ainsi pour n'importe quel autre évènement du cycle de vie, ce qui inclut :

- `preRemove`
- `postRemove`
- `prePersist`
- `postPersist`
- `preUpdate`
- `postUpdate`
- `postLoad`
- `loadClassMetadata`

Pour plus d'informations sur la signification de ces évènements du cycle de vie et sur leurs callbacks en général, référez-vous à la documentation de Doctrine: *Lifecycle Events documentation*[10].

10. `http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/events.html#lifecycle-events`

Callbacks du cycle de vie et écouteurs d'évènements

Notez que la méthode `setCreatedValue()` ne prend pas d'argument. C'est toujours le cas des callbacks du cycle de vie, et c'est intentionnel : ces callbacks doivent être de simples méthodes et contiennent des transformations de données internes à l'entité (ex: définir un champ créé ou mis à jour, générer une valeur de slug...).

Si vous souhaitez faire des montages plus lourds - comme une identification ou envoyer un mail - vous devez écrire une classe externe et l'enregistrer pour écouter ou s'abonner aux évènements, puis lui donner les accès à toutes les ressources dont vous aurez besoin. Pour plus d'informations, lisez *Comment enregistrer des listeners (« écouteurs » en français) et des souscripteurs d'évènement*.

Les extensions de Doctrine: Timestampable, Sluggable, etc.

Doctrine est très flexible, et il existe un certain nombre d'extensions tierces qui permettent de faciliter les tâches courantes sur vos entités. Elles incluent diverses outils comme *Sluggable*, *Timestampable*, *Loggable*, *Translatable*, et *Tree*.

Pour plus d'informations sur comment trouver et utiliser ces extensions, regardez l'article du cookbook relatif à l'*utilisation des extensions Doctrine*.

Référence des types de champs de Doctrine

Doctrine contient un grand nombre de types de champs. Chacun mappe un type de données PHP vers un type de colonne spécifique à la base de données que vous utilisez. Les types suivants sont supportés par Doctrine :

- **Chaînes de caractères**
  - `string` (utilisé pour des chaînes courtes)
  - `text` (utilisé pour des chaînes longues)
- **Nombres**
  - `integer`
  - `smallint`
  - `bigint`
  - `decimal`
  - `float`
- **Dates et heures** (ces champs utilisent un objet PHP *DateTime*[11])
  - `date`
  - `time`
  - `datetime`
- **Autre types**
  - `boolean`
  - `object` (sérialisé et stocké dans un champ `CLOB`)
  - `array` (sérialisé et stocké dans un champ `CLOB`)

Pour plus d'informations, lisez la documentation Doctrine *Types de mapping Doctrine*[12].

11. `http://php.net/manual/en/class.datetime.php`

Options des champs

Un ensemble d'options peut être appliqué à chaque champ. Les options disponibles incluent `type` (valant `string` par défaut), `name`, `length`, `unique` et `nullable`. Regardons quelques exemples :

Listing 8-47
```php
/**
 * Une chaîne de caractères de longueur 255 qui ne peut pas être nulle
 * (reflétant les valeurs par défaut des options "type", "length" et *nullable);
 *
 * @ORM\Column()
 */
protected $name;

/**
 * Une chaîne de longueur 150 qui sera persistée vers une colonne "email_address"
 * et a un index unique.
 *
 * @ORM\Column(name="email_address", unique=true, length=150)
 */
protected $email;
```

Il existe d'autre options qui ne sont pas listées ici. Pour plus de détails, lisez *Property Mapping documentation*[13].

Commandes en console

L'intégration de l'ORM Doctrine2 offre plusieurs commandes de console sous l'espace de nom `doctrine`. Pour voir la liste de ces commandes, vous pouvez exécutez la console sans aucun argument :

Listing 8-48
```
$ php app/console
```

Une liste des commandes disponibles s'affichera, la plupart d'entre elles commencent par le préfixe `doctrine:`. Vous pouvez obtenir plus d'informations sur n'importe laquelle de ces commandes (ou n'importe quelle commande Symfony) en lançant la commande `help`. Par exemple, pour obtenir des informations sur la commande `doctrine:database:create`, exécutez :

Listing 8-49
```
$ php app/console help doctrine:database:create
```

Voici une liste non exhaustive de commandes intéressantes :

- `doctrine:ensure-production-settings` - teste si l'environnement actuel est configuré de manière optimale pour la production. Elle devrait toujours être exécutée dans un environnement *prod* :

Listing 8-50
```
$ php app/console doctrine:ensure-production-settings --env=prod
```

- `doctrine:mapping:import` - permet à Doctrine d'introspecter une base de données existante pour créer les informations de mapping. Pour plus d'informations, voir *Comment générer des Entités à partir d'une base de données existante*.

12. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#doctrine-mapping-types
13. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#property-mapping

- `doctrine:mapping:info` - vous donne toutes les entités dont Doctrine a connaissance et s'il existe des erreurs basiques dans leur mapping.
- `doctrine:query:dql` et `doctrine:query:sql` - vous permet d'effectuer des commandes DQL ou SQL directement en ligne de commande.

Pour pouvoir charger des données d'installation (fixtures), vous devrez installer le bundle `DoctrineFixtureBundle`. Pour apprendre comment le faire, lisez le chapitre du Cookbook : "*DoctrineFixturesBundle*"

Cette page montre comment Doctrine fonctionne au sein d'un contrôleur. Mais vous voulez peut être travailler avec Doctrine ailleurs dans votre application. La méthode *`getDoctrine()`*[14] de la classe Controller retourne le service `doctrine`. Vous pouvez travailler avec de la même manière, en l'injectant dans vos propres services. Lisez *Service Container* pour savoir comment créer vos propres services.

Résumé

Avec Doctrine, vous pouvez tout d'abord vous focaliser sur vos objets et sur leur utilité dans votre application, puis vous occuper de leur persistance ensuite. Vous pouvez faire cela car Doctrine vous permet d'utiliser n'importe quel objet PHP pour stocker vos données et se fie aux métadonnées de mapping pour faire correspondre les données d'un objet à une table particulière de la base de données.

Et même si Doctrine tourne autour d'un simple concept, il est incroyablement puissant, vous permettant de créer des requêtes complexes et de vous abonner à des évènements qui vous permettent d'effectuer différentes actions au cours du cycle de vie de vos objets.

Pour plus d'informations sur Doctrine, lisez la section *Doctrine* du Cookbook: *cookbook*, qui inclut les articles suivants :

- *DoctrineFixturesBundle*
- *Comment utiliser les extensions Doctrine: Timestampable, Sluggable, Translatable, etc.*

14. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#getDoctrine()

Chapter 9
Propel et les bases de données

Regardons la vérité en face, l'une des tâches les plus communes et difficiles pour quelconque application implique de devoir persister et lire de l'information dans et depuis une base de données. Symfony2 ne vient pas avec un ORM intégré, mais l'intégration de Propel reste néanmoins aisée. Pour commencer, lisez *Travailler Avec Symfony2*[1].

Un Exemple Simple : Un Produit

Dans cette section, vous allez configurer votre base de données, créer un objet `Product`, le persister dans la base de données et ensuite le récupérer.

Codez en même temps que vous lisez cet exemple

Si vous voulez codez en même temps que vous parcourez l'exemple de ce chapitre, créez un `AcmeStoreBundle` via :

Listing 9-1
```
$ php app/console generate:bundle --namespace=Acme/StoreBundle
```

Configurer la Base de Données

Avant de pouvoir démarrer, vous allez avoir besoin de configurer les informations de connexion de votre base de données. Par convention, ces informations sont généralement configurées dans un fichier `app/config/parameters.yml` :

Listing 9-2
```
# app/config/parameters.yml
parameters:
    database_driver:   mysql
    database_host:     localhost
    database_name:     test_project
```

1. http://www.propelorm.org/cookbook/symfony2/working-with-symfony2.html#installation

```
6       database_user:     root
7       database_password: password
8       database_charset:  UTF8
```

Définir la configuration via `parameters.yml` n'est qu'une convention. Les paramètres définis dans ce fichier sont référencés par le fichier de la configuration principale lorsque vous définissez Propel :

Listing 9-3
```
1  propel:
2      dbal:
3          driver:     "%database_driver%"
4          user:       "%database_user%"
5          password:   "%database_password%"
6          dsn:
   "%database_driver%:host=%database_host%;dbname=%database_name%;charset=%database_charset%"
```

Maintenant que Propel connaît votre base de données, Symfony2 peut créer cette dernière pour vous :

Listing 9-4
```
1  $ php app/console propel:database:create
```

Dans cet exemple, vous avez une connexion configurée, nommée `default`. Si vous voulez configurer plus d'une connexion, lisez la partie sur la configuration <Travailler avec Symfony2 - Configuration>`_.

Créer une Classe Modèle

Dans le monde de Propel, les classes ActiveRecord sont connues en tant que **modèles** car les classes générées par Propel contiennent un peu de logique métier.

Pour les gens qui utilisent Symfony2 avec Doctrine2, les **modèles** sont équivalents aux **entités**.

Supposons que vous construisiez une application dans laquelle des produits doivent être affichés. Tout d'abord, créez un fichier `schema.xml` dans le répertoire `Resources/config` de votre `AcmeStoreBundle` :

Listing 9-5
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <database name="default" namespace="Acme\StoreBundle\Model" defaultIdMethod="native">
3      <table name="product">
4          <column name="id" type="integer" required="true" primaryKey="true"
5  autoIncrement="true" />
6          <column name="name" type="varchar" primaryString="true" size="100" />
7          <column name="price" type="decimal" />
8          <column name="description" type="longvarchar" />
9      </table>
   </database>
```

Construire le Modèle

Après avoir créé votre `schema.xml`, générez votre modèle à partir de ce dernier en exécutant :

Listing 9-6
```
$ php app/console propel:model:build
```

Cela va générer chaque classe modèle afin que vous puissiez développer rapidement votre application dans le répertoire `Model/` de votre bundle `AcmeStoreBundle`.

Créer les Tables et le Schéma de la Base de Données

Maintenant, vous avez une classe `Product` utilisable et tout ce dont vous avez besoin pour persister un produit. Bien sûr, pour le moment, vous ne disposez pas de la table `product` correspondante dans votre base de données. Heureusement, Propel peut automatiquement créer toutes les tables de base de données nécessaires à chaque modèle connu de votre application. Pour effectuer cela, exécutez :

Listing 9-7
```
$ php app/console propel:sql:build
$ php app/console propel:sql:insert --force
```

Votre base de données possède désormais une table `product` entièrement fonctionnelle avec des colonnes qui correspondent au schéma que vous avez spécifié.

Vous pouvez exécuter les trois dernières commandes de manière combinée en utilisant la commande suivante : `php app/console propel:build --insert-sql`.

Persister des Objets dans la Base de Données

Maintenant que vous avez un objet `Product` et une table `product` correspondante, vous êtes prêt à persister des données dans la base de données. Depuis un contrôleur, cela est assez facile. Ajoutez la méthode suivante au `DefaultController` du bundle créé plus haut:

Listing 9-8
```
// src/Acme/StoreBundle/Controller/DefaultController.php
use Acme\StoreBundle\Model\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

public function createAction()
{
    $product = new Product();
    $product->setName('A Foo Bar');
    $product->setPrice(19.99);
    $product->setDescription('Lorem ipsum dolor');

    $product->save();

    return new Response('Created product id '.$product->getId());
}
```

Dans ce bout de code, vous instanciez et travaillez avec l'objet `product`. Lorsque vous appelez la méthode `save()` sur ce dernier, vous persistez le produit dans la base de données. Pas besoin d'utiliser d'autres services, l'objet sait comment se persister lui-même.

Si vous codez tout en lisant cet exemple, vous allez avoir besoin de créer une *route* qui pointe vers cette action pour la voir fonctionner.

Récupérer des Objets depuis la Base de Données

Récupérer un objet depuis la base de données est encore plus simple. Par exemple, supposons que vous ayez configuré une route pour afficher un `Produit` spécifique basé sur la valeur de son `id`:

Listing 9-9

```
use Acme\StoreBundle\Model\ProductQuery;

public function showAction($id)
{
    $product = ProductQuery::create()
        ->findPk($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    // faites quelque chose, comme passer l'objet $product à un template
}
```

Mettre à jour un Objet

Une fois que vous avez récupéré un objet depuis Propel, le mettre à jour est facile. Supposons que vous ayez une route qui fasse correspondre un « id » de produit à une action de mise à jour dans un contrôleur:

Listing 9-10

```
use Acme\StoreBundle\Model\ProductQuery;

public function updateAction($id)
{
    $product = ProductQuery::create()
        ->findPk($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    $product->setName('New product name!');
    $product->save();

    return $this->redirect($this->generateUrl('homepage'));
}
```

Mettre à jour un objet implique seulement trois étapes :

1. récupérer l'objet depuis Propel ;
2. modifier l'objet ;
3. le sauvegarder.

Supprimer un Objet

Supprimer un objet est très similaire, excepté que cela requiert un appel à la méthode `delete()` sur l'objet:

Listing 9-11
```php
$product->delete();
```

Effectuer des requêtes pour récupérer des Objets

Propel fournit des classes `Query` générées pour exécuter aussi bien des requêtes basiques que des requêtes complexes, et cela sans aucun effort de votre part:

Listing 9-12
```php
\Acme\StoreBundle\Model\ProductQuery::create()->findPk($id);

\Acme\StoreBundle\Model\ProductQuery::create()
    ->filterByName('Foo')
    ->findOne();
```

Imaginez que vous souhaitiez effectuer une requête sur des produits coûtant plus de 19.99, ordonnés du moins cher au plus cher. Depuis l'un de vos contrôleurs, faites ce qui suit:

Listing 9-13
```php
$products = \Acme\StoreBundle\Model\ProductQuery::create()
    ->filterByPrice(array('min' => 19.99))
    ->orderByPrice()
    ->find();
```

Très facilement, vous obtenez vos produits en construisant une requête de manière orientée objet. Aucun besoin de perdre du temps à écrire du SQL ou quoi que ce soit d'autre, Symfony2 offre une manière de programmer totalement orientée objet et Propel respecte la même philosophie en fournissant une couche d'abstraction ingénieuse.

Si vous voulez réutiliser certaines requêtes, vous pouvez ajouter vos propres méthodes à la classe `ProductQuery`:

Listing 9-14
```php
// src/Acme/StoreBundle/Model/ProductQuery.php

class ProductQuery extends BaseProductQuery
{
    public function filterByExpensivePrice()
    {
        return $this
            ->filterByPrice(array('min' => 1000))
    }
}
```

Mais notez que Propel génère beaucoup de méthodes pour vous et qu'une simple méthode `findAllOrderedByName()` peut être écrite sans aucun effort:

Listing 9-15
```php
\Acme\StoreBundle\Model\ProductQuery::create()
    ->orderByName()
    ->find();
```

Relations/Associations

Supposez que les produits de votre application appartiennent tous à une seule « catégorie ». Dans ce cas, vous aurez besoin d'un objet `Category` et d'une manière de lier un objet `Product` à un objet `Category`.

Commencez par ajouter la définition de `category` dans votre `schema.xml` :

Listing 9-16
```xml
<database name="default" namespace="Acme\StoreBundle\Model" defaultIdMethod="native">
    <table name="product">
        <column name="id" type="integer" required="true" primaryKey="true"
autoIncrement="true" />
        <column name="name" type="varchar" primaryString="true" size="100" />
        <column name="price" type="decimal" />
        <column name="description" type="longvarchar" />

        <column name="category_id" type="integer" />
        <foreign-key foreignTable="category">
            <reference local="category_id" foreign="id" />
        </foreign-key>
    </table>

    <table name="category">
        <column name="id" type="integer" required="true" primaryKey="true"
autoIncrement="true" />
        <column name="name" type="varchar" primaryString="true" size="100" />
    </table>
</database>
```

Créez les classes :

Listing 9-17
```
$ php app/console propel:model:build
```

Assumons que vous ayez des produits dans votre base de données, vous ne souhaitez pas les perdre. Grâce aux migrations, Propel va être capable de mettre à jour votre base de données sans perdre aucune donnée.

Listing 9-18
```
$ php app/console propel:migration:generate-diff
$ php app/console propel:migration:migrate
```

Votre base de données a été mise à jour, vous pouvez continuer à écrire votre application.

Sauvegarder des Objets Liés

Maintenant, voyons le code en action. Imaginez que vous soyez dans un contrôleur:

Listing 9-19
```php
// ...
use Acme\StoreBundle\Model\Category;
use Acme\StoreBundle\Model\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

class DefaultController extends Controller
{
    public function createProductAction()
    {
        $category = new Category();
        $category->setName('Main Products');

        $product = new Product();
        $product->setName('Foo');
        $product->setPrice(19.99);
        // lie ce produit à la catégorie définie plus haut
```

```
        $product->setCategory($category);

        // sauvegarde l'ensemble
        $product->save();

        return new Response(
            'Created product id: '.$product->getId().' and category id:
'.$category->getId()
        );
    }
}
```

Maintenant, une seule ligne est ajoutée aux deux tables `category` et `product`. La colonne `product.category_id` pour le nouveau produit est définie avec l'id de la nouvelle catégorie. Propel gère la persistance de cette relation pour vous.

Récupérer des Objets Liés

Lorsque vous avez besoin de récupérer des objets liés, votre processus ressemble à la récupération d'un attribut dans l'exemple précédent. Tout d'abord, récupérez un objet `$product` et ensuite accédez à sa `Category` liée:

Listing 9-20

```
// ...
use Acme\StoreBundle\Model\ProductQuery;

public function showAction($id)
{
    $product = ProductQuery::create()
        ->joinWithCategory()
        ->findPk($id);

    $categoryName = $product->getCategory()->getName();

    // ...
}
```

Notez que dans l'exemple ci-dessus, seulement une requête a été effectuée.

Plus d'informations sur les Associations

Vous trouverez plus d'informations sur les relations en lisant le chapitre dédié sur les *Relations*[2].

Callbacks et Cycles de Vie

De temps en temps, vous avez besoin d'effectuer une action juste avant ou juste après qu'un objet soit inséré, mis à jour, ou supprimé. Ces types d'actions sont connus en tant que callbacks de « cycles de vie » ou « hooks », comme ce sont des méthodes callbacks que vous devez exécuter à différentes étapes du cycle de vie d'un objet (par exemple : l'objet est inséré, mis à jour, supprimé, etc).

Pour ajouter un « hook », ajoutez simplement une nouvelle méthode à la classe de l'objet:

Listing 9-21

2. http://www.propelorm.org/documentation/04-relationships.html

```php
// src/Acme/StoreBundle/Model/Product.php

// ...

class Product extends BaseProduct
{
    public function preInsert(\PropelPDO $con = null)
    {
        // faites quelque chose avant que l'objet soit inséré
    }
}
```

Propel fournit les « hooks » suivants :

- `preInsert()` code exécuté avant l'insertion d'un nouvel objet
- `postInsert()` code exécuté après l'insertion d'un nouvel objet
- `preUpdate()` code exécuté avant la mise à jour d'un objet existant
- `postUpdate()` code exécuté après la mise à jour d'un objet existant
- `preSave()` code exécuté avant la sauvegarde d'un objet (nouveau ou existant)
- `postSave()` code exécuté après la sauvegarde d'un objet (nouveau ou existant)
- `preDelete()` code exécuté avant la suppression d'un objet
- `postDelete()` code exécuté après la suppression d'un objet

Comportements

Tous les comportements fournis par Propel fonctionnent avec Symfony2. Pour avoir plus d'informations sur la manière d'utiliser les comportements Propel, jetez un oeil à la *Section de Référence des Comportements*[3].

Commandes

Vous devriez lire la section dédiée aux *commandes Propel dans Symfony2*[4].

3. http://www.propelorm.org/documentation/#behaviors_reference
4. http://www.propelorm.org/cookbook/symfony2/working-with-symfony2#the_commands

Chapter 10
Les Tests

A chaque fois que vous écrivez une nouvelle ligne de code, vous ajoutez aussi potentiellement de nouveaux bugs. Pour créer de meilleures applications et plus fiables, vous devriez tester votre code avec des tests fonctionnels et des tests unitaires.

Le Framework de Test PHPUnit

Symfony2 intègre une bibliothèque indépendante - appelée PHPUnit - qui vous fournit un framework de tests complet. Ce chapitre ne couvre par PHPUnit elle-même, mais elle a sa propre *documentation*[1].

Symfony2 fonctionne avec PHPUnit 3.5.11 au minimum, mais la version 3.6.4 est nécessaire pour tester le coeur du framework.

Chaque test - que ce soit un test fonctionnel ou un test unitaire - est une classe PHP qui devrait se trouver dans le sous-répertoire *Tests/* de vos bundles. Si vous suivez cette règle, alors vous pouvez lancer les tests de votre application avec la commande suivante :

Listing 10-1
```
# spécifier le répertoire de configuration
$ phpunit -c app/
```

L'option `-c` indique à PHPUnit de chercher le fichier de configuration dans le répertoire `app/`. Si vous voulez en savoir plus sur les options de PHPUnit, jetez un oeil au fichier `app/phpunit.xml.dist`.

La couverture de code peut être générée avec l'option `--coverage-html`.

1. `http://www.phpunit.de/manual/3.5/en/`

Tests unitaires

Un test unitaire teste habituellement une classe PHP spécifique. Si vous voulez tester le comportement général de votre application, lisez la section sur les Tests Fonctionnels.

Écrire des tests unitaires avec Symfony2 n'est pas différent d'écrire des tests unitaires standards PHPUnit. Supposez, par exemple, que vous avez un classe *incroyablement* simple appelée `Calculator` dans le répertoire `Utility/` de votre bundle :

Listing 10-2
```php
// src/Acme/DemoBundle/Utility/Calculator.php
namespace Acme\DemoBundle\Utility;

class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

Pour tester cela, créez le fichier `CalculatorTest` dans le dossier `Tests/Utility` de votre:

Listing 10-3
```php
// src/Acme/DemoBundle/Tests/Utility/CalculatorTest.php
namespace Acme\DemoBundle\Tests\Utility;

use Acme\DemoBundle\Utility\Calculator;

class CalculatorTest extends \PHPUnit_Framework_TestCase
{
    public function testAdd()
    {
        $calc = new Calculator();
        $result = $calc->add(30, 12);

        // vérifie que votre classe a correctement calculé!
        $this->assertEquals(42, $result);
    }
}
```

Par convention, le sous-répertoire `Tests/` doit reproduire la structure de votre bundle. Donc, si vous testez une classe du répertoire `Utility/` du bundle, mettez le test dans le répertoire `Tests/Utility/`.

Comme dans votre vraie application, le chargement automatique est activé via le fichier `bootstrap.php.cache` (comme configuré par défaut dans le fichier `phpunit.xml.dist`).

Exécuter les tests pour un fichier ou répertoire donné est aussi très facile :

Listing 10-4
```
# lancer tous les tests du répertoire Utility
$ phpunit -c app src/Acme/DemoBundle/Tests/Utility/

# lancer les tests de la classe Calculator
$ phpunit -c app src/Acme/DemoBundle/Tests/Utility/CalculatorTest.php

# lancer tous les tests d'un Bundle
$ phpunit -c app src/Acme/DemoBundle/
```

Tests Fonctionnels

Les tests fonctionnels vérifient l'intégration des différentes couches d'une application (du routing jusqu'aux vues). Ils ne sont pas différents des tests unitaires tant que PHPUnit est concerné, mais ils possèdent un workflow très spécifique :

- Faire une requête;
- Tester la réponse;
- Cliquer sur un lien ou soumettre un formulaire;
- Tester la réponse;
- Recommencer, ainsi de suite.

Votre premier test fonctionnel

Les tests fonctionnels sont de simples fichiers PHP qui se trouvent dans le répertoire `Tests/Controller` du bundle. Si vous voulez tester des pages gérées par la classe `DemoController`, commencez par créer un nouveau fichier `DemoControllerTest.php` qui étend la classe spéciale `WebTestCase`.

Par exemple, la Symfony2 Standard Edition fournit un simple test fonctionnel pour son `DemoController` (*DemoControllerTest*[2]) :

Listing 10-5
```php
// src/Acme/DemoBundle/Tests/Controller/DemoControllerTest.php
namespace Acme\DemoBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DemoControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();

        $crawler = $client->request('GET', '/demo/hello/Fabien');

        $this->assertGreaterThan(0, $crawler->filter('html:contains("Hello
Fabien")')->count());
    }
}
```

Pour lancer vos tests fonctionnels, la classe `WebTestCase` initie (bootstrap) le noyau (kernel) de votre application. Dans la plupart des cas, cela fonctionnera automatiquement. Cependant, si votre kernel n'est pas dans le répertoire habituel, vous devrez modifier le fichier `phpunit.xml.dist` pour définir la variable d'environnement `KERNEL_DIR` afin qu'elle pointe vers le répertoire du kernel :

Listing 10-6
```xml
<phpunit>
    <!-- ... -->
    <php>
        <server name="KERNEL_DIR" value="/path/to/your/app/" />
    </php>
    <!-- ... -->
</phpunit>
```

2. https://github.com/symfony/symfony-standard/blob/master/src/Acme/DemoBundle/Tests/Controller/DemoControllerTest.php

La méthode `createClient()` retourne un client, qui ressemble au navigateur que vous utilisez pour surfer sur vos sites:

Listing 10-7
```php
$crawler = $client->request('GET', '/demo/hello/Fabien');
```

La méthode `request()` (plus d'informations sur *la méthode Request*) retourne un objet *`Crawler`*[3] qui peut être utilisé pour sélectionner les éléments dans la Réponse, cliquer sur les liens, et soumettre des formulaires.

Le Crawler peut être utilisé seulement si le contenu de la Réponse est un document XML ou HTML. Pour obtenir le contenu brut, appelez la méthode `$client->getResponse()->getContent()`.

Cliquez sur un lien en le sélectionnant avec le Crawler en utilisant soit une expression XPath ou un sélecteur CSS, puis utilisez le Client pour cliquer dessus. Par exemple, le code suivant trouve tous les liens avec le text `Greet`, puis sélectionne le second et clique dessus :

Listing 10-8
```php
$link = $crawler->filter('a:contains("Greet")')->eq(1)->link();

$crawler = $client->click($link);
```

Soumettre un formulaire est très similaire : sélectionnez un bouton de ce dernier, réécrivez quelques-unes de ses valeurs si besoin est puis soumettez-le :

Listing 10-9
```php
$form = $crawler->selectButton('submit')->form();

// définit certaines valeurs
$form['name'] = 'Lucas';
$form['form_name[subject]'] = 'Hey there!';

// soumet le formulaire
$crawler = $client->submit($form);
```

Le formulaire peut également gérer les envois et contient des méthodes pour remplir les différents types des champs de formulaire (ex `select()` et `tick()`). Pour plus de détails, lisez la section Formulaires.

Maintenant que vous pouvez naviguer aisément à travers une application, utilisez les assertions pour tester qu'elle fait effectivement ce que vous souhaitez qu'elle fasse. Utilisez le Crawler pour faire des assertions sur le DOM :

Listing 10-10
```php
// affirme que la réponse correspond au sélecteur CSS donné
$this->assertGreaterThan(0, $crawler->filter('h1')->count());
```

Ou alors, testez directement le contenu de la Réponse si vous voulez juste vérifier qu'il contient un certain texte, ou si la Réponse n'est pas un document XML/HTML :

Listing 10-11
```php
$this->assertRegExp('/Hello Fabien/', $client->getResponse()->getContent());
```

3. http://api.symfony.com/master/Symfony/Component/DomCrawler/Crawler.html

Plus d'infos sur la méthode `request()`:

La signature complète de la méthdode `request()` est:

Listing 10-12
```php
request(
    $method,
    $uri,
    array $parameters = array(),
    array $files = array(),
    array $server = array(),
    $content = null,
    $changeHistory = true
)
```

Le tableau `server` contient les valeurs brutes que vous trouveriez normalement dans la variable superglobale *$_SERVER*[4]. Par exemple, pour définir les entêtes HTTP *Content-Type* et *Referer*, vous procéderiez comme suit (pensez au préfixe *HTTP_* pour les en-têtes non standard) :

Listing 10-13
```php
$client->request(
    'GET',
    '/demo/hello/Fabien',
    array(),
    array(),
    array(
        'CONTENT_TYPE' => 'application/json',
        'HTTP_REFERER' => '/foo/bar',
        'HTTP_X-Requested-With' => 'XMLHttpRequest',
    )
);
```

4. http://php.net/manual/en/reserved.variables.server.php

Assertions Utiles

Afin que vous démarriez plus rapidement, voici une liste des assertions les plus communes et utiles :

Listing 10-14
```php
// Vérifie qu'il y a au moins une balise h2 dans la classe "subtitle"
$this->assertGreaterThan(0, $crawler->filter('h2.subtitle')->count());

// Vérifie qu'il y a exactement 4 balises h2 sur la page
$this->assertCount(4, $crawler->filter('h2'));

// Vérifie que l'entête "Content-Type" vaut "application/json"
$this->assertTrue($client->getResponse()->headers->contains('Content-Type',
'application/json'));

// Vérifie que le contenu retourné correspond à la regex
$this->assertRegExp('/foo/', $client->getResponse()->getContent());

// Vérifie que le status de la réponse est 2xx
$this->assertTrue($client->getResponse()->isSuccessful());
// Vérifie que le status de la réponse est 404
$this->assertTrue($client->getResponse()->isNotFound());
// Vérifie un status spécifique
$this->assertEquals(200, $client->getResponse()->getStatusCode());

// Vérifie que la réponse est redirigée vers /demo/contact
$this->assertTrue($client->getResponse()->isRedirect('/demo/contact'));
// ou vérifie simplement que la réponse est redirigée vers une URL quelconque
$this->assertTrue($client->getResponse()->isRedirect());
```

Travailler avec le Client Test

Le Client Test simule un client HTTP comme un navigateur et lance des requêtes à votre application Symfony2 :

Listing 10-15
```php
$crawler = $client->request('GET', '/hello/Fabien');
```

La méthode `request()` prend en arguments la méthode HTTP et une URL, et retourne une instance de `Crawler`.

Utilisez le Crawler pour trouver des éléments DOM dans la Réponse. Ces éléments peuvent ainsi être utilisés pour cliquer sur des liens et soumettre des formulaires :

Listing 10-16
```php
$link = $crawler->selectLink('Go elsewhere...')->link();
$crawler = $client->click($link);

$form = $crawler->selectButton('validate')->form();
$crawler = $client->submit($form, array('name' => 'Fabien'));
```

Les méthodes `click()` et `submit()` retournent toutes deux un objet `Crawler`. Ces méthodes sont le meilleur moyen de naviguer dans une application, car elles s'occupent de beaucoup de choses pour vous, comme détecter la méthode HTTP à partir d'un formulaire et vous fournir une bonne API pour uploader des fichiers.

Vous en apprendrez davantage sur les objets `Link` et `Form` dans la section *Crawler* ci-dessous.

Mais vous pouvez aussi simuler les soumissions de formulaires et des requêtes complexes grâce à la méthode `request()` :

Listing 10-17
```php
// Soumettre directement un formulaire (mais utiliser le Crawler est plus simple)
$client->request('POST', '/submit', array('name' => 'Fabien'));

// Soumission de formulaire avec upload de fichier
use Symfony\Component\HttpFoundation\File\UploadedFile;

$photo = new UploadedFile(
    '/path/to/photo.jpg',
    'photo.jpg',
    'image/jpeg',
    123
);
// or
$photo = array(
    'tmp_name' => '/path/to/photo.jpg',
    'name' => 'photo.jpg',
    'type' => 'image/jpeg',
    'size' => 123,
    'error' => UPLOAD_ERR_OK
);
$client->request(
    'POST',
    '/submit',
    array('name' => 'Fabien'),
    array('photo' => $photo)
);

// Exécute une requête DELETE et passe des entête HTTP
$client->request(
    'DELETE',
    '/post/12',
    array(),
    array(),
    array('PHP_AUTH_USER' => 'username', 'PHP_AUTH_PW' => 'pa$$word')
);
```

Enfin, vous pouvez forcer chaque requête à être exécutée dans son propre processus PHP afin d'éviter quelconques effets secondaires quand vous travaillez avec plusieurs clients dans le même script :

Listing 10-18
```php
$client->insulate();
```

Naviguer

Le Client supporte de nombreuses opérations qui peuvent être effectuées à travers un navigateur réel :

Listing 10-19
```php
$client->back();
$client->forward();
$client->reload();
```

```
4
5 // efface tous les cookies et l'historique
6 $client->restart();
```

Accéder aux Objets Internes

Si vous utilisez le client pour tester votre application, vous pourriez vouloir accéder aux objets internes du client :

Listing 10-20
```
1 $history   = $client->getHistory();
2 $cookieJar = $client->getCookieJar();
```

Vous pouvez aussi obtenir les objets liés à la dernière requête :

Listing 10-21
```
1 $request  = $client->getRequest();
2 $response = $client->getResponse();
3 $crawler  = $client->getCrawler();
```

Si vos requêtes ne sont pas isolées, vous pouvez aussi accéder au `Container` et au `Kernel` :

Listing 10-22
```
1 $container = $client->getContainer();
2 $kernel    = $client->getKernel();
```

Accéder au Container

Il est fortement recommandé qu'un test fonctionnel teste seulement la Réponse. Mais dans certains cas rares, vous pourriez vouloir accéder à quelques objets internes pour écrire des assertions. Dans tels cas, vous pouvez accéder au conteneur d'injection de dépendances :

Listing 10-23
```
1 $container = $client->getContainer();
```

Notez bien que cela ne fonctionne pas si vous isolez le client ou si vous utilisez une couche HTTP. Pour une liste des services disponibles dans votre application, utilisez la commande `container:debug`.

Si les informations que vous avez besoin de vérifier sont disponibles via le profileur, utilisez le.

Accéder aux données du profileur

Pour chaque requête, vous pouvez activer le profileur Symfony afin de collecter et stocker diverses informations sur la gestion interne des requêtes. Par exemple, le profileur peut être utilisé pour vérifier qu'une page donnée ne dépasse pas un certain nombre de requêtes en base de données lors de son chargement.

Pour obtenir le profileur de la dernière requête, utilisez le code suivant:

Listing 10-24
```
1 // active le profileur pour la requête suivante
2 $client->enableProfiler();
3
```

```
4  $crawler = $client->request('GET', '/profiler');
5
6  // récupère le profil
7  $profile = $client->getProfile();
```

Pour des détails spécifique sur l'utilisation du profileur au sein d'un test, lisez la documentation du cookbook *Comment utiliser le Profiler dans un test fonctionnel*.

Redirection

Lorsqu'une requête retourne une réponse redirigée, le client ne la suit pas automatiquement. Vous pouvez examiner la réponse puis forcer une redirection grâce à la méthode `followRedirect()`:

Listing 10-25
```
1  $crawler = $client->followRedirect();
```

Si vous voulez que le client suive automatiquement tous les redirections, vous pouvez le forcer avec la méthode `followRedirects()`:

Listing 10-26
```
1  $client->followRedirects();
```

Le Crawler

Une instance de Crawler est retournée chaque fois que vous effectuez une requête avec le Client. Elle vous permet de naviguer à travers des documents HTML, de sélectionner des noeuds, de trouver des liens et des formulaires.

Traverser

Comme jQuery, le Crawler possède des méthodes lui permettant de naviguer à travers le DOM d'un document HTML/XML. Par exemple, le code suivant trouve tout les éléments `input[type=submit]`, sélectionne le dernier de la page et sélectionne son élément parent immédiat :

Listing 10-27
```
1  $newCrawler = $crawler->filter('input[type=submit]')
2      ->last()
3      ->parents()
4      ->first()
5  ;
```

Beaucoup d'autres méthodes sont également disponibles :

Method	Description
`filter('h1.title')`	Noeuds qui correspondent au sélecteur CSS
`filterXpath('h1')`	Noeuds qui correspondent à l'expression XPath
`eq(1)`	Noeud pour l'index spécifié
`first()`	Premier noeud
`last()`	Dernier noeud
`siblings()`	Éléments de même niveau (siblings)

Method	Description
`nextAll()`	Tous les siblings suivants
`previousAll()`	Tous les siblings précédents
`parents()`	Retourne les noeuds parents
`children()`	Retourne les noeuds enfants
`reduce($lambda)`	Noeuds pour lesquels $lambda ne retourne pas false

Puisque chacune de ses méthodes retourne une instance de `Crawler`, vous pouvez affiner votre sélection de noeuds en enchainant les appels de méthodes :

Listing 10-28
```php
$crawler
    ->filter('h1')
    ->reduce(function ($node, $i)
    {
        if (!$node->getAttribute('class')) {
            return false;
        }
    })
    ->first();
```

Utilisez la fonction `count()` pour avoir le nombre de noeuds contenus dans le Crawler : `count($crawler)`

Extraction d'informations

Le Crawler peut extraire des informations des noeuds :

Listing 10-29
```php
// retourne la valeur de l'attribut du premier noeud
$crawler->attr('class');

// retourne la valeur du noeud pour le premier noeud
$crawler->text();

// extrait un tableau d'attributs pour tous les noeuds (_text retourne
// la valeur du noeud)
// retourne un tableau de chaque élément du crawler, chacun avec les attributs
// value et href
$info = $crawler->extract(array('_text', 'href'));

// exécute une lambda pour chaque noeud et retourne un tableau de résultats
$data = $crawler->each(function ($node, $i)
{
    return $node->attr('href');
});
```

Liens

Vous pouvez sélectionner les liens grâce aux méthodes ci-dessus ou grâce au raccourci très pratique `selectLink()` :

Listing 10-30
```
$crawler->selectLink('Click here');
```

Cela sélectionne tous les liens qui contiennent le texte donné, ou les images cliquables dont l'attribut `alt``contient ce texte. Comme les autres méthodes de filtre, cela retourne un autre objet ``Crawler`.

Une fois que vous avez sélectionné un lien, vous avez accès à l'objet spécial `Link`, qui possède des méthodes utiles et spécifiques aux liens (comme `getMethod()` et `getUri()`).Pour cliquer sur un lien, utilisez la méthode `click()` du Client et passez la à un objet `Link` :

Listing 10-31
```
$link = $crawler->selectLink('Click here')->link();

$client->click($link);
```

Formulaires

Comme pour les liens, vous sélectionnez les formulaires à l'aide de la méthode `selectButton()` :

Listing 10-32
```
$buttonCrawlerNode = $crawler->selectButton('submit');
```

Notez que vous sélectionnez les boutons de formulaire et non pas les formulaires eux-mêmes, car un formulaire peut contenir plusieurs boutons; si vous utilisez l'API de traversement, gardez en mémoire que vous devez chercher un bouton.

La méthode `selectButton()` peut sélectionner des balises `button` et des balises `input` de type submit; elle utilise plusieurs parties du bouton pour les trouver :

- La valeur de l'attribut `value`;
- La valeur de l'attribut `id` ou `alt` pour les images;
- La valeur de l'attribut `id` ou `name` pour les balises `button`.

Lorsque vous avez un Crawler qui représente un bouton, appelez la méthode `form()` pour obtenir une instance de `Form` pour le formulaire contenant le noeud du bouton :

Listing 10-33
```
$form = $buttonCrawlerNode->form();
```

Quand vous appelez la méthode `form()`, vous pouvez aussi passer un tableau de valeurs de champs qui réécrit les valeurs par défaut :

Listing 10-34
```
$form = $buttonCrawlerNode->form(array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

Et si vous voulez simuler une méthode HTTP spécifique pour le formulaire, passez la comme second argument :

Listing 10-35
```
$form = $buttonCrawlerNode->form(array(), 'DELETE');
```

Le Client peut soumettre des instances de `Form` :

Listing 10-36

```php
$client->submit($form);
```

Les valeurs des champs peuvent aussi être passées en second argument de la méthode `submit()` :

Listing 10-37
```php
$client->submit($form, array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

Pour les situations plus complexes, utilisez l'instance de `Form` en tant que tableau pour définir la valeur de chaque champ individuellement :

Listing 10-38
```php
// Change la valeur d'un champ
$form['name'] = 'Fabien';
$form['my_form[subject]'] = 'Symfony rocks!';
```

Il y a aussi une sympathique API qui permet de manipuler les valeurs des champs selon leur type :

Listing 10-39
```php
// sélectionne une option/radio
$form['country']->select('France');

// Coche une checkbox
$form['like_symfony']->tick();

// Upload un fichier
$form['photo']->upload('/path/to/lucas.jpg');
```

Vous pouvez obtenir les valeurs qui sont soumises en appelant la méthode `getValues()` de l'objet `Form`. Les fichiers uploadés sont disponibles dans un tableau séparé retourné par `getFiles()`. Les méthodes `getPhpValues()` et `getPhpFiles()` retournent aussi les valeurs soumises, mais au format PHP (cela convertit les clés avec la notation entre crochets - ex `my_form[subject]` - en tableaux PHP).

Configuration de Test

Le Client utilisé par les tests fonctionnels crée un Kernel qui est exécuté dans un environnement de `test`. Puisque Symfony charge le `app/config/config_test.yml` dans l'environnement de `test`, vous pouvez modifier la configuration de votre application spécifiquement pour les tests.

Par exemple, par défaut, le swiftmailer est configuré pour ne *pas* envoyer de mails en environnement de `test`. Vous pouvez le voir sous l'option de configuration `swiftmailer` :

Listing 10-40
```yaml
# app/config/config_test.yml
# ...

swiftmailer:
    disable_delivery: true
```

Vous pouvez aussi utiliser un environnement entièrement différent, ou surcharger le mode debug par défaut (`true`) en les passant en tant qu'option à la méthode `createClient()` :

Listing 10-41

```php
$client = static::createClient(array(
    'environment' => 'my_test_env',
    'debug'       => false,
));
```

Si votre application se comporte selon certaines en-têtes HTTP, passez-les en tant que second argument de `createClient()` :

Listing 10-42
```php
$client = static::createClient(array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

Vous pouvez aussi réécrire les en-têtes HTTP par requête (i.e. et non pas pour toutes les requêtes) :

Listing 10-43
```php
$client->request('GET', '/', array(), array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

Le client test est disponible en tant que service dans le conteneur en environnement de `test` (ou n'importe où si l'option *framework.test* est activée). Cela signifie que vous pouvez surcharge entièrement le service en cas de besoin.

Configuration PHPUnit

Chaque application possède sa propre configuration PHPUnit, stockée dans le fichier `phpunit.xml.dist`. Vous pouvez éditer ce fichier pour changer les valeurs par défaut ou vous pouvez créer un fichier `phpunit.xml` pour personnaliser la configuration de votre machine locale.

Stockez le fichier `phpunit.xml.dist` dans votre gestionnaire de code, et ignorez le fichier `phpunit.xml`.

Par défaut, seulement les tests situés dans des bundles « standards » sont exécutés par la commande `phpunit` (standard étant des tests situés dans les répertoires `src/*/Bundle/Tests` ou `src/*/Bundle/*Bundle/Tests`). Mais vous pouvez aisément ajouter d'autres répertoires. Par exemple, la configuration suivante ajoute les tests de bundles tiers installés :

Listing 10-44
```xml
<!-- hello/phpunit.xml.dist -->
<testsuites>
    <testsuite name="Project Test Suite">
        <directory>../src/*/*Bundle/Tests</directory>
        <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
    </testsuite>
</testsuites>
```

Pour inclure d'autres répertoires dans la couverture du code, éditez aussi la section `<filter>` :

Listing 10-45
```xml
<filter>
    <whitelist>
```

```
        <directory>../src</directory>
        <exclude>
            <directory>../src/*/*Bundle/Resources</directory>
            <directory>../src/*/*Bundle/Tests</directory>
            <directory>../src/Acme/Bundle/*Bundle/Resources</directory>
            <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
        </exclude>
    </whitelist>
</filter>
```

En savoir plus

- *Le Composant DomCrawler*
- *Le Composant CssSelector*
- *Comment simuler une authentification HTTP dans un Test Fonctionnel*
- *Comment tester les interactions de multiples clients*
- *Comment utiliser le Profiler dans un test fonctionnel*
- *Comment personnaliser le processus de bootstrap avant les tests*

Chapter 11
Validation

La validation est une tâche très commune dans les applications web. Les données saisies dans les formulaires doivent être validées. Les données doivent également être validées avant d'être écrites dans une base de données ou transmises à un service Web.

Symfony2 est livré avec un composant *Validator*[1] qui rend cette tâche facile et transparente. Ce composant est basé sur la *spécification JSR303*[2]. Quoi ? Une spécification Java en PHP? Vous avez bien entendu, mais ce n'est pas aussi terrible que ça en a l'air. Regardons comment elle peut être utilisée en PHP.

Les Bases de la Validation

La meilleure façon de comprendre la validation est de la voir en action. Pour commencer, supposons que vous avez créé un bon vieil objet PHP que vous avez besoin d'utiliser quelque part dans votre application :

Listing 11-1
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    public $name;
}
```

Jusqu'à présent, c'est juste une classe ordinaire qui est utile dans votre application. L'objectif de la validation est de vous dire si oui ou non les données d'un objet sont valides. Pour que cela fonctionne, vous allez configurer une liste de règles (appelée *constraints*) que l'objet doit respecter pour être valide. Ces règles peuvent être spécifiées via un certain nombre de formats (YAML, XML, les annotations, ou PHP).

Par exemple, pour garantir que la propriété `$name` ne soit pas vide, ajoutez le code suivant :

Listing 11-2

1. https://github.com/symfony/Validator
2. http://jcp.org/en/jsr/detail?id=303

```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        name:
            - NotBlank: ~
```

Les propriétés protégées et privées peuvent également être validées, ainsi que les méthodes « getter » (voir validator-constraint-targets).

Utiliser le Service `validator`

Ensuite, pour vraiment valider un objet `Author`, utilisez la méthode `validate` du service validator (class *`Validator`*[3]). Le travail du `validator` est simple : lire les contraintes (règles) d'une classe et vérifier si les données sur l'objet satisfont ces contraintes ou non. Si la validation échoue, un tableau d'erreurs est retourné. Prenez cet exemple simple d'une action d'un contrôleur :

Listing 11-3

```php
use Symfony\Component\HttpFoundation\Response;
use Acme\BlogBundle\Entity\Author;
// ...

public function indexAction()
{
    $author = new Author();
    // ... do something to the $author object

    $validator = $this->get('validator');
    $errorList = $validator->validate($author);

    if (count($errorList) > 0) {
        return new Response(print_r($errorList, true));
    } else {
        return new Response('The author is valid! Yes!');
    }
}
```

Si la propriété `$name` est vide, vous allez voir le message d'erreur suivant :

Listing 11-4

```
Acme\BlogBundle\Author.name:
    This value should not be blank
```

Si vous insérez une valeur dans la propriété `name`, le message de succès va apparaître.

La plupart du temps, vous n'aurez pas à interagir directement avec le service `validator` ou besoin de vous inquiéter concernant l'affichage des erreurs. La plupart du temps, vous allez utiliser la validation indirectement lors de la soumission des données du formulaire. Pour plus d'informations, consultez-le *Validation et Formulaires*.

Vous pouvez aussi passer une collection d'erreurs à un template.

Listing 11-5

3. http://api.symfony.com/master/Symfony/Component/Validator/Validator.html

```php
if (count($errorList) > 0) {
    return $this->render('AcmeBlogBundle:Author:validate.html.twig', array(
        'errorList' => $errorList,
    ));
} else {
    // ...
}
```

A l'intérieur d'un template, vous pouvez afficher la liste des erreurs comme vous voulez :

Listing 11-6
```twig
{# src/Acme/BlogBundle/Resources/views/Author/validate.html.twig #}

<h3>The author has the following errors</h3>
<ul>
{% for error in errorList %}
    <li>{{ error.message }}</li>
{% endfor %}
</ul>
```

Chaque erreur de validation (appelée une « violation de contrainte »), est représentée par un objet *ConstraintViolation*[4].

Validation et Formulaires

Le service `validator` peut être utilisé à tout moment pour valider n'importe quel objet. En réalité, vous travaillerez le plus souvent avec le `validator` indirectement lorsque vous utilisez les formulaires. La bibliothèque de formulaires de Symfony utilise le service `validator` en interne pour valider l'objet après que les valeurs ont été soumises. Les violations de contraintes sur l'objet sont converties en objets `FieldError` qui peuvent être facilement affichés dans votre formulaire. Le processus de soumission d'un formulaire typique ressemble au code suivant:

Listing 11-7
```php
// ...
use Acme\BlogBundle\Entity\Author;
use Acme\BlogBundle\Form\AuthorType;
use Symfony\Component\HttpFoundation\Request;

public function updateAction(Request $request)
{
    $author = new Author();
    $form = $this->createForm(new AuthorType(), $author);

    if ($request->isMethod('POST')) {
        $form->bind($request);

        if ($form->isValid()) {
            // the validation passed, do something with the $author object

            return $this->redirect($this->generateUrl(...));
        }
    }

```

4. http://api.symfony.com/master/Symfony/Component/Validator/ConstraintViolation.html

```
21      return $this->render('BlogBundle:Author:form.html.twig', array(
22          'form' => $form->createView(),
23      ));
24  }
```

Cet exemple utilise une classe de formulaire `AuthorType`, qui n'est pas montrée ici.

Pour plus d'informations, voir le chapitre *Forms*.

Configuration

Le validateur Symfony2 est activé par défaut, mais vous devez activer explicitement les annotations, si vous voulez les utiliser pour spécifier vos contraintes :

Listing 11-8
```
1  # app/config/config.yml
2  framework:
3      validation: { enable_annotations: true }
```

Contraintes

Le `validator` est conçu pour valider des objets selon des *contraintes* (règles). Afin de valider un objet, il suffit d'associer une ou plusieurs contraintes à sa classe et ensuite de le passer au service `validator`.

Dans les coulisses, une contrainte est simplement un objet PHP qui déclare une règle. Dans la vraie vie, une contrainte pourrait être : « Le gâteau ne doit pas être brûlé ». En Symfony2, les contraintes sont similaires : ce sont des garanties qu'une condition est vraie. Pour une valeur donnée, une contrainte vous dira si cette valeur obéit aux règles ou non.

Contraintes supportées

Symfony2 est fourni avec un grand nombre des contraintes les plus utiles habituellement.

Contraintes de bases

Voici les contraintes de base : utilisez les afin de vérifier des données basiques à propos de la valeur d'une propriété ou de la valeur retournée par une méthode de votre objet.

- *NotBlank*
- *Blank*
- *NotNull*
- *Null*
- *True*
- *False*
- *Type*

Contraintes sur les chaînes de caractères

- *Email*

- *Length*
- *Url*
- *Regex*
- *Ip*

Contraintes sur les nombres

- *Range*

Contraintes comparatives

- *EqualTo*
- *NotEqualTo*
- *IdenticalTo*
- *NotIdenticalTo*
- *LessThan*
- *LessThanOrEqual*
- *GreaterThan*
- *GreaterThanOrEqual*

Contraintes sur les dates

- *Date*
- *DateTime*
- *Time*

Contraintes sur les collections

- *Choice*
- *Collection*
- *Count*
- *UniqueEntity*
- *Language*
- *Locale*
- *Country*

Contraintes sur les fichiers

- *File*
- *Image*

Contraintes sur la finance et autres numéros

- *CardScheme*
- *Luhn*
- *Iban*
- *Isbn*
- *Issn*

Autres Contraintes

- *Callback*
- *All*

- *UserPassword*
- *Valid*

Vous pouvez aussi créer vos propres contraintes. Ce sujet est abordé dans l'article «*Comment créer une Contrainte de Validation Personnalisée*» du cookbook.

Configuration de contraintes

Certaines contraintes, comme *NotBlank*, sont simples alors que d'autres, comme la contrainte *Choice* ont plusieurs options de configuration disponibles. Supposons que la classe `Author` a une autre propriété, `gender` qui peut prendre comme valeur « homme » ou « femme » :

Listing 11-9
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: { choices: [homme, femme], message: Choose a valid gender. }
```

Les options d'une contrainte peuvent toujours être passées en tant que tableau. Certaines contraintes, cependant, vous permettent également de passer la valeur d'une option « *par défaut* » au lieu du tableau. Dans le cas de la contrainte `Choice`, les options de `choices` peuvent être spécifiées de cette manière.

Listing 11-10
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: [homme, femme]
```

Ceci a pour unique but de rendre la configuration de l'option la plus probable plus courte et plus rapide.

Si jamais vous n'êtes pas certain de la façon dont spécifier une option, soit vérifiez la documentation de l'API pour la contrainte concernée, soit jouez la sécurité en passant toujours dans un tableau d'options (la première méthode indiquée ci-dessus).

Traduction des messages de contrainte

Pour des informations sur la traduction des messages de contrainte, allez voir *Traduire les messages de contraintes*.

Objectifs des contraintes

Les contraintes peuvent être appliquées à une propriété de classe (par ex. `name`) ou une méthode publique getter (par ex. `getFullName`). La première méthode est la plus commune et facile à utiliser, mais la seconde vous permet de spécifier des règles de validation plus complexes.

Propriétés

Valider des propriétés de classe est la technique de validation la plus basique. Symfony2 vous permet de valider des propriétés privées, protégées ou publiques. Le prochain exemple vous montre comment configurer la propriété `$firstName` d'une classe `Author` pour avoir au moins 3 caractères.

Listing 11-11
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
```

```
4           firstName:
5               - NotBlank: ~
6               - MinLength:
7                                       min: 3
```

Getters

Les contraintes peuvent également être appliquées à la valeur de retour d'une méthode. Symfony2 vous permet d'ajouter une contrainte à toute méthode publique dont le nom commence par « get » ou « is ». Dans ce guide, ces deux types de méthodes sont désignées comme « getters ».

L'avantage de cette technique est qu'elle vous permet de valider votre objet dynamiquement. Par exemple, supposons que vous voulez vous assurer qu'un champ mot de passe ne correspond pas au prénom de l'utilisateur (pour des raisons de sécurité). Vous pouvez le faire en créant une méthode `isPasswordLegal`, puis en spécifiant que cette méthode doit retourner `true` :

Listing 11-12
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    getters:
        passwordLegal:
            - "True": { message: "Le mot de passe et le prénom doivent être différents" }
```

Maintenant, créez la méthode `isPasswordLegal()`, et faites ce dont vous avez besoin:

Listing 11-13
```php
public function isPasswordLegal()
{
    return ($this->firstName != $this->password);
}
```

Les plus perspicaces d'entre vous auront remarqué que le préfixe du getter (« get » ou « is ») est omis dans le mapping. Cela vous permet de déplacer la contrainte sur une propriété du même nom plus tard (ou vice versa) sans changer votre logique de validation.

Classes

Certaines contraintes s'appliquent à toute la classe qui est validée. Par exemple, la contrainte *Callback* est une contrainte générique qui s'applique à la classe elle-même. Lorsque cette classe est validée, les méthodes spécifiées par cette contrainte sont simplement exécutées pour personnaliser encore plus la validation

Groupes de Validation

Jusqu'ici, vous avez été en mesure d'ajouter des contraintes à une classe et demander si oui ou non cette classe satisfait toutes les contraintes définies. Dans certains cas, cependant, vous aurez besoin de valider un objet en ne prenant en compte que *certaines* des contraintes de la classe. Pour ce faire, vous pouvez organiser chaque contrainte en un ou plusieurs « groupes de validation », et ensuite appliquer la validation sur un groupe de contraintes seulement.

Par exemple, supposons que vous avez une classe `User`, qui est utilisée à la fois quand un utilisateur s'enregistre et quand un utilisateur met à jour son profil plus tard :

Listing 11-14

```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\User:
    properties:
        email:
            - Email: { groups: [registration] }
        password:
            - NotBlank: { groups: [registration] }
            - Length: { min: 7, groups: [registration] }
        city:
            - Length:
                                    min: 2
```

Avec cette configuration, il y a deux groupes de validation :

- `Default` - contient les contraintes non affectées à tout autre groupe ;
- `registration` - contient les contraintes sur les champs `email` and `password` seulement.

Pour dire au validateur d'utiliser un groupe spécifique, passer un ou plusieurs noms de groupe comme deuxième argument de la méthode `validate()`

Listing 11-15
```php
$errorList = $validator->validate($author, array('registration'));
```

Bien sûr, vous travaillerez bien souvent indirectement avec la validation via la bibliothèque de formulaire. Pour plus d'informations sur la façon d'utiliser les groupes de validation à l'intérieur des formulaires, voir *Les Groupes de Validation*.

Valider des valeurs et des tableaux

Jusqu'ici, vous avez vu comment valider des objets entiers. Mais souvent, vous voudrez juste valider une simple valeur, comme par exemple vérifier qu'une chaîne de caractères est une adresse email valide. C'est en fait très facile à faire. Dans un contrôleur, ça ressemble à ceci:

Listing 11-16
```php
// Ajouter ceci en haut de votre classe
use Symfony\Component\Validator\Constraints\Email;

public function addEmailAction($email)
{
    $emailConstraint = new Email();
    // toutes les "options" de contrainte sont définies comme suit
    $emailConstraint->message = 'Invalid email address';

    // utiliser le validator pour valider la valeur
    $errorList = $this->get('validator')->validateValue($email, $emailConstraint);

    if (count($errorList) == 0) {
        // c'est une adresse valide, faire quelque chose
    } else {
        // ce n'est *pas* une adresse valide
        $errorMessage = $errorList[0]->getMessage()

        // faire quelque chose avec l'erreur
    }

    // ...
}
```

En appelant `validateValue` du validator, vous pouvez passer une valeur brute et l'objet contrainte dont vous voulez valider la valeur. Une liste complète des contraintes disponibles - ainsi que le nom de classe complet de chaque contrainte - est disponible dans le chapitre *contraintes*.

La méthode `validateValue` retourne un objet *`ConstraintViolationList`*[5] qui se comporte comme un tableau d'erreurs. Chaque erreur de la collection est un objet *`ConstraintViolation`*[6], qui contient le message d'erreur dans sa méthode *getMessage*.

Le mot de la fin

Le `validator` de Symfony2 est un outil puissant qui peut être un atout pour garantir que les données de n'importe quel objet sont « valides ». La puissance de la validation réside dans les « contraintes », qui sont des règles que vous pouvez appliquer aux propriétés ou aux méthodes getter de votre objet. Et tandis que vous utiliserez plus communément le système de validation indirectement lors de l'utilisation des formulaires, n'oubliez pas qu'il peut être utilisé partout pour valider n'importe quel objet.

En savoir plus avec le Cookbook

- *Comment créer une Contrainte de Validation Personnalisée*

5. http://api.symfony.com/master/Symfony/Component/Validator/ConstraintViolationList.html
6. http://api.symfony.com/master/Symfony/Component/Validator/ConstraintViolation.html

Chapter 12
Formulaires

Intéragir avec les formulaires HTML est l'une des tâches les plus habituelles - et stimulantes - pour un développeur web. Symfony2 intègre un composant « Form » qui rend cette intéraction très facile. Dans ce chapitre, vous allez construire un formulaire complexe depuis la base, tout en apprenant au fur et à mesure les caractéristiques les plus importantes de la bibliothèque des formulaires.

Le composant formulaire de Symfony est une bibliothèque autonome qui peut être utilisée en dehors des projets Symfony2. Pour plus d'informations, voir le *Composant Formulaire Symfony2*[1] sur Github.

Créer un formulaire simple

Supposez que vous construisiez une application « todo list » (en français : « liste de choses à faire ») simple qui doit afficher des « tâches ». Parce que vos utilisateurs devront éditer et créer des tâches, vous allez avoir besoin de construire des formulaires. Mais avant de commencer, concentrez-vous d'abord sur la classe générique `Task` qui représente et stocke les données pour une tâche:

Listing 12-1
```php
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

class Task
{
    protected $task;

    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }
    public function setTask($task)
```

1. https://github.com/symfony/Form

```
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }
    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

Si vous codez en même temps que vous lisez cet exemple, créez en premier le bundle `AcmeTaskBundle` en exécutant la commande suivante (et en acceptant toutes les options par défaut):

Listing 12-2
```
$ php app/console generate:bundle --namespace=Acme/TaskBundle
```

Cette classe est un « bon vieil objet PHP » parce que, jusqu'ici, elle n'a rien à voir avec Symfony ou un quelconque autre bibliothèque. C'est tout simplement un objet PHP normal qui répond directement à un besoin dans *votre* application (c-a-d le besoin de représenter une tâche dans votre application). Bien sûr, à la fin de ce chapitre, vous serez capable de soumettre des données à une instance de `Task` (via un formulaire HTML), de valider ses données, et de les persister dans la base de données.

Construire le Formulaire

Maintenant que vous avez créé une classe `Task`, la prochaine étape est de créer et d'afficher le formulaire HTML. Dans Symfony2, cela se fait en construisant un objet formulaire qui est ensuite affiché dans un template. Pour l'instant, tout ceci peut être effectué dans un contrôleur:

Listing 12-3
```
// src/Acme/TaskBundle/Controller/DefaultController.php
namespace Acme\TaskBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Acme\TaskBundle\Entity\Task;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function newAction(Request $request)
    {
        // crée une tâche et lui donne quelques données par défaut pour cet exemple
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', 'text')
            ->add('dueDate', 'date')
            ->add('save', 'submit')
            ->getForm();

```

```
23          return $this->render('AcmeTaskBundle:Default:new.html.twig', array(
24              'form' => $form->createView(),
25          ));
26      }
27  }
```

Cet exemple vous montre comment construire votre formulaire directement depuis le contrôleur. Plus tard, dans la section « *Créer des Classes de Formulaire* », vous apprendrez à construire votre formulaire dans une classe autonome, ce qui est recommandé, car vos formulaires deviendront ainsi réutilisables.

Créer un formulaire nécessite relativement peu de code, car les objets formulaires de Symfony2 sont construits avec un « constructeur de formulaire » (« form builder »). Le principe du constructeur de formulaire est de vous permettre d'écrire des « conteneurs » de formulaire simples, et de le laisser prendre en charge la plus grosse partie du travail qu'est la construction du formulaire.

Dans cet exemple, vous avez ajouté deux champs à votre formulaire - `task` et `dueDate` - correspondants aux propriétés `task` et `dueDate` de votre classe `Task`. Vous avez aussi assigné à chaque champ un « type » (par exemple : `text`, `date`), qui, entre autres, détermine quelle(s) balise(s) HTML est affichée pour ce champ. Enfin, vous avez ajouté un bouton « submit » pour envoyer le formulaire sur le serveur.

New in version 2.3: Le support des boutons « submit » est une nouveauté de Symfony 2.3. Avant cela, vous deviez ajouter vous même les boutons au formulaire HTML.

Symfony2 est livré avec de nombreux types prédéfinis qui seront présentés rapidement plus tard (voir *Types de champ intégrés*).

Affichage du Formulaire

Maintenant que le formulaire a été créé, la prochaine étape est de l'afficher. Ceci est fait en passant un objet spécial « vue de formulaire » à votre template (notez le `$form->createView()` dans le contrôleur ci-dessus) et en utilisant un ensemble de fonctions d'aide (helpers) pour les formulaires :

Listing 12-4
```
1  {# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}
2
3  {{ form(form) }}
```

Cet exemple suppose que vous soumettez le formulaire grâce à une requête « POST », et à la même URL que celle qui l'a affiché. Vous apprendrez plus tard comment changer la méthode de la requête et l'URL cible du formulaire.

C'est tout ! En affichant `form(form)`, chaque champ du formulaire est affiché, avec un label et un message d'erreur (si erreur il y a). La fonction `form` affiche tout dans une balise HTML `form`. Aussi facile que cela soit, ce n'est pas (encore) très flexible. Habituellement, vous voudrez afficher chaque champ du formulaire individuellement afin de pouvoir contrôler ce à quoi le formulaire ressemble. Vous apprendrez comment faire cela dans la section «*Afficher un Formulaire dans un Template*».

Avant de continuer, notez comment le champ `task` affiché possède la valeur de la propriété `task` de l'objet `$task` (c-a-d « Write a blog post »). C'est le premier travail d'un formulaire : de prendre les données d'un objet et de les traduire dans un format adapté pour être affiché dans un formulaire HTML.

Le système de formulaire est assez intelligent pour accéder la valeur de la propriété protégée `task` via les méthodes `getTask()` et `setTask()` de la classe `Task`. A moins qu'une propriété soit publique, elle *doit* avoir une méthode « getter » et un « setter » afin que le composant formulaire puisse récupérer et assigner des données à cette propriété. Pour une propriété booléenne, vous pouvez utiliser une méthode « isser » ou « hasser » (par exemple : `isPublished()` ou `hasReminder()`) à la place d'un getter (par exemple : ``getPublished()` ou `getReminder()`).

Gérer la Soumission des Formulaires

Le second travail d'un formulaire est de transmettre les données soumises par l'utilisateur aux propriétés d'un objet. Pour réaliser cela, les données soumises par l'utilisateur doivent être liées au formulaire. Ajoutez la fonctionnalité suivante à votre contrôleur:

Listing 12-5

```php
// ...
use Symfony\Component\HttpFoundation\Request;

public function newAction(Request $request)
{
    // just setup a fresh $task object (remove the dummy data)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', 'text')
        ->add('dueDate', 'date')
        ->add('save', 'submit')
        ->getForm();

    $form->handleRequest($request);

    if ($form->isValid()) {
        // fait quelque chose comme sauvegarder la tâche dans la bdd

        return $this->redirect($this->generateUrl('task_success'));
    }

    // ...
}
```

New in version 2.3: La méthode *`handleRequest()`*[2] a été ajoutée dans Symfony 2.3. Avant cela, la `$request` était passée à la méthode `submit`, une stratégie qui est dépréciée et qui sera supprimée dans Symfony 3.0. Pour plus de détails sur cette méthode, lisez *cookbook-form-submit-request*.

2. http://api.symfony.com/master/SymfonyComponentFormFormInterface.html#handleRequest()

Ce contrôleur suit un pattern commun dans la manière de gérer les formulaires, et a trois scénarios possibles :

1. Lors du chargement initial de la page dans votre navigateur, le formulaire est simplement créé et affiché. *`handleRequest()`*[3] détermine que le formulaire n'a pas été soumis et ne fait rien. *`isValid()`*[4] retourne `false` si le formulaire n'a pas été soumis.
2. Lorsque l'utilisateur soumet le formulaire, *`handleRequest()`*[5] s'en rend compte et écrit immédiatement les données soumises dans les propriétés `task` et `dueDate` de l'objet `$task`. Ensuite, cet objet est validé. S'il est invalide (la validation est abordée dans la section suivante), *`isValid()`*[6] retourne encore `false` et le formulaire est affiché de nouveau avec toutes ses erreurs de validation.

Vous pouvez utiliser la méthode *`isSubmitted()`*[7] pour vérifier si un formulaire a été soumis ou non, indépendemment du fait que les données soumises soient valides ou non.

3. Lorsque l'utilisateur soumet le formulaire avec des données valides, les données soumises sont toujours écrites dans l'objet, mais cette fois, *`isValid()`*[8] retourne `true`. Vous avez alors la possibilité d'effectuer certaines actions qui utilisent l'objet ``$task` (comme le persister en base de données) avant de rediriger l'utilisateur vers une autre page (par exemple une page « merci » ou « ça a marché ! »).

Rediriger un utilisateur après une soumission de formulaire réussie empêche l'utilisateur de pouvoir rafraichir la page et de resoumettre les données.

Submitting Forms with Multiple Buttons

New in version 2.3: Le support des boutons dans les formulaires est une nouveauté de Symfony 2.3.

Lorsque votre formulaire contient plus d'un bouton submit, vous devrez vérifier quel bouton a été utilisé pour adapter les actions à effectuer dans votre contrôleur. Ajoutons un second bouton avec l'intitulé « Enregistrer et nouveau » (save and add) à notre formulaire:

Listing 12-6
```php
$form = $this->createFormBuilder($task)
    ->add('task', 'text')
    ->add('dueDate', 'date')
    ->add('save', 'submit')
    ->add('saveAndAdd', 'submit')
    ->getForm();
```

Dans votre contrôleur, utilisez la méthode *`isClicked()`*[9] du bouton pour vérifier si un clic sur le bouton « Enregistrer et nouveau » a été fait:

3. http://api.symfony.com/master/SymfonyComponentFormFormInterface.html#handleRequest()
4. http://api.symfony.com/master/SymfonyComponentFormFormInterface.html#isValid()
5. http://api.symfony.com/master/SymfonyComponentFormFormInterface.html#handleRequest()
6. http://api.symfony.com/master/SymfonyComponentFormFormInterface.html#isValid()
7. http://api.symfony.com/master/SymfonyComponentFormFormInterface.html#isSubmitted()
8. http://api.symfony.com/master/SymfonyComponentFormFormInterface.html#isValid()
9. http://api.symfony.com/master/Symfony/Component/Form/ClickableInterface.html#isClicked()

Listing 12-7
```php
if ($form->isValid()) {
    // fait quelque chose comme sauvegarder la tâche dans la bdd

    $nextAction = $form->get('saveAndAdd')->isClicked()
        ? 'task_new'
        : 'task_success';

    return $this->redirect($this->generateUrl($nextAction));
}
```

Validation de formulaire

Dans la section précédente, vous avez appris comment un formulaire peut être soumis avec des données valides ou non. Dans Symfony2, la validation est appliquée à l'objet sous-jacent (par exemple : `Task`). En d'autres termes, la question n'est pas de savoir si le « formulaire » est valide, mais plutôt de savoir si l'objet `$task` est valide ou non après que le formulaire lui a appliqué les données. Appeler `$form->isValid()` est un raccourci qui demande à l'objet `$task` si oui ou non il possède des données valides.

La validation est effectuée en ajoutant un ensemble de règles (appelées contraintes) à une classe. Pour voir cela en action, ajoutez des contraintes de validation afin que le champ `task` ne puisse pas être vide et que le champ `dueDate` ne puisse pas être vide et qu'il doive être un objet DateTime valide.

Listing 12-8
```yaml
# Acme/TaskBundle/Resources/config/validation.yml
Acme\TaskBundle\Entity\Task:
    properties:
        task:
            - NotBlank: ~
        dueDate:
            - NotBlank: ~
            - Type: \DateTime
```

C'est tout ! Si vous soumettez le formulaire à nouveau avec des données non valides, vous allez voir les erreurs correspondantes affichées avec le formulaire.

Validation HTML5

Grâce à HTML5, de nombreux navigateurs peuvent nativement forcer certaines contraintes de validation côté client. La validation la plus commune est activée en affichant un attribut `required` sur les champs qui sont requis. Pour les navigateurs qui supportent HTML5, cela entrainera l'affichage d'un message natif du navigateur si l'utilisateur essaye de soumettre le formulaire avec ce champ vide.

Les formulaires générés profitent pleinement de cette nouvelle fonctionnalité en ajoutant des attributs HTML qui déclenchent la validation. La validation côté client, néanmoins, peut être désactivée en ajoutant l'attribut `novalidate` à la balise `form` ou `formnovalidate` à la balise `submit`. Ceci est particulièrement utile quand vous voulez tester vos contraintes de validation côté serveur, mais que vous en êtes empêché par le formulaire de votre navigateur, par exemple, quand vous soumettez des champs vides.

La validation est une fonctionnalité très puissante de Symfony2 et possède son *propre chapitre*.

Les Groupes de Validation

Si vous n'utilisez pas *les groupes de validation*, alors vous pouvez sauter cette section.

Si votre objet profite des avantages des *groupes de validation*, vous aurez besoin de spécifier quel(s) groupe(s) de validation votre formulaire doit utiliser:

Listing 12-9
```php
$form = $this->createFormBuilder($users, array(
    'validation_groups' => array('registration'),
))->add(...);
```

Si vous créez *des classes de formulaire* (une bonne pratique), alors vous devrez ajouter ce qui suit à la méthode `setDefaultOptions()`:

Listing 12-10
```php
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array('registration'),
    ));
}
```

Dans les deux cas, le groupe de validation `registration` *uniquement* sera utilisé pour valider l'objet sous-jacent.

Désactiver la validation

New in version 2.3: La possibilité de définir `validation_groups` à false a été ajoutée dans Symfony 2.3, bien que la définir comme un tableau vide revient au même dans les versions précédentes.

Parfois, il peut être utilise de supprimer complètement la validation d'un formulaire. Pour cela, vous pouvez éviter l'appel à la méthode ***isValid()***[10] dans votre contrôleur. Si cela n'est pas possible, vous pouvez également définir l'option `validation_groups` à `false` ou en tant que tableau vide:

Listing 12-11
```php
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => false,
    ));
}
```

Notez que lorsque vous faites ceci, le formulaire exécutera toujours des vérifications de bases, comme par exemple véirifier si un fichier est trop gros pour être uploadé, ou si des champs qui n'existent pas sont soumis. Si vous voulez supprimer complètement la validation, supprimez l'appel à la méthode ***isValid()***[11] dans votre contrôleur.

10. http://api.symfony.com/master/Symfony/Component/Form/FormInterface.html#isValid()
11. http://api.symfony.com/master/Symfony/Component/Form/FormInterface.html#isValid()

Groupes basés sur les données soumises

Si vous avez besoin de plus de logique pour déterminer les groupes de validation (c'est-à-dire en vous basant sur les données), vous pouvez définir l'option `validation_groups` comme un tableau callback:

Listing 12-12
```php
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array('Acme\\AcmeBundle\\Entity\\Client',
'determineValidationGroups'),
        'validation_groups' => array(
            'Acme\AcmeBundle\Entity\Client',
            'determineValidationGroups',
        ),
    ));

}
```

Cela appellera la méthode statique `determineValidationGroups()` de la classe `Client` après que le formulaire est soumis, mais avant que la validation ne soit faite. L'objet Form est passé comme argument à cette méthode (regardez l'exemple suivant). Vous pouvez aussi définir une logique entière en utilisant une Closure:

Listing 12-13
```php
use Symfony\Component\Form\FormInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => function(FormInterface $form) {
            $data = $form->getData();
            if (Entity\Client::TYPE_PERSON == $data->getType()) {
                return array('person');
            } else {
                return array('company');
            }
        },
    ));
}
```

Groupes basés sur les clics de boutons

New in version 2.3: Le support des boutons dans les formulaires est une nouveauté de Symfony 2.3.

Lorsque votre formulaire contient plusieurs boutons submit, vous pouvez changer de groupe de validation selon le bouton qui a été utilisé pour soumettre le formulaire. Par exemple, considérez un formulaire d'assistance qui vous permet d'aller à une étape suivante, ou de revenir à une étape précédente. Supposez également que lorsque vous retournez à l'étape précédente, les données du formulaires doivent être enregistrées mais pas validées.

Premièrement, vous avez besoin d'ajouter les deux boutons au formulaire:

Listing 12-14
```php
$form = $this->createFormBuilder($task)
    // ...
    ->add('nextStep', 'submit')
    ->add('previousStep', 'submit')
    ->getForm();
```

Ensuite, configurez le bouton qui renvoie à l'étape précédente pour qu'il exécute des groupes de validation spécifiques. Dans cet exemple, vous voulez qu'il supprime la validation, donc vous devez définir ses options `validation_groups` à false:

Listing 12-15
```php
$form = $this->createFormBuilder($task)
    // ...
    ->add('previousStep', 'submit', array(
        'validation_groups' => false,
    ))
    ->getForm();
```

Maintenant, le formulaire ne vérifiera pas les contraintes de validation. Mais il effectuera toujours des vérifications d'intégrité de base, comme vérifier si un fichier est trop gros pour être uploadé ou si vous tentez de soumettre un texte dans un champ censé contenir un nombre.

Types de champ intégrés

Symfony est fourni par défaut avec un grand nombre de types de champ qui couvrent la plupart des champs de formulaire existants et des types de données que vous pourrez rencontrer :

Champs de type Texte

- *text*
- *textarea*
- *email*
- *integer*
- *money*
- *number*
- *password*
- *percent*
- *search*
- *url*

Champs de type Choice

- *choice*
- *entity*
- *country*
- *language*
- *locale*
- *timezone*
- *currency*

Champs de type Date et Time

- *date*

- *datetime*
- *time*
- *birthday*

Autres champs

- *checkbox*
- *file*
- *radio*

Champs de type Groupe

- *collection*
- *repeated*

Champs cachés

- *hidden*

Bouttons

- *button*
- *reset*
- *submit*

Champs de base

- *form*

Vous pouvez aussi créer vos propres types de champ personnalisés. Ce sujet est couvert par l'article du cookbook « *Comment Créer un Type de Champ de Formulaire Personnalisé* ».

Options des types de champ

Chaque type de champ possède un nombre d'options qui peuvent être utilisées pour le configurer. Par exemple, le champ `dueDate` est affiché par défaut comme 3 champs select. Cependant, le *champ date* peut être configuré pour être affiché en tant qu'un unique champ texte (où l'utilisateur doit entrer la date « manuellement » comme une chaîne de caractères):

Listing 12-16
```
1  ->add('dueDate', 'date', array('widget' => 'single_text'))
```

Chaque type de champ a un certain nombre d'options différentes qui peuvent lui être passées. Beaucoup d'entre elles sont spécifiques à chacun d'entres-eux et vous pouvez trouver ces détails dans la documentation de chaque type.

L'option `required`

La plus commune des options est l'option `required`, qui peut être appliquée à tous les champs. Par défaut, cette dernière est définie à `true`, signifiant que les navigateurs supportant HTML5 vont appliquer la validation côté client si le champ est laissé vide. Si vous ne souhaitez pas ce comportement, vous pouvez soit définir l'option `required` de vos champs à `false`, ou soit *désactiver la validation HTML5*.

Notez aussi que définir l'option `required` à `true` **ne** créera **aucune** validation côté serveur. En d'autres termes, si un utilisateur soumet une valeur vide pour le champ (soit avec un vieux navigateur ou via un service web, par exemple), cette valeur sera acceptée comme valide à moins que vous n'utilisiez la contrainte de validation de Symfony `NotBlank` ou `NotNull`.

En d'autres termes, l'option `required` est « cool », mais une réelle validation côté serveur devrait *toujours* être mise en place.

The `label` option

Le label d'un champ de formulaire peut être défini grâce à l'option `label`, qui s'applique à n'importe quel champ:

Listing 12-17
```
->add('dueDate', 'date', array(
    'widget' => 'single_text',
    'label'  => 'Due Date',
))
```

Le label d'un champ peut aussi être défini dans le template lorsque vous affichez le formulaire, voir ci-dessous.

Prédiction de Type de Champ

Maintenant que vous avez ajouté les métadonnées de validation à la classe `Task`, Symfony en sait déjà un peu plus à propos de vos champs. Si vous l'autorisez, Symfony peut « prédire » le type de vos champs et les mettre en place pour vous. Dans cet exemple, Symfony peut deviner depuis les règles de validation que le champ `task` est un champ `texte` classique et que `dueDate` est un champ `date`:

Listing 12-18
```
public function newAction()
{
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task')
        ->add('dueDate', null, array('widget' => 'single_text'))
        ->getForm();
}
```

La « prédiction » est activée lorsque vous omettez le second argument de la méthode `add()` (ou si vous lui passez `null`). Si vous passez un tableau d'options en tant que troisième argument (comme pour `dueDate` ci-dessus), ces options sont appliquées au champ prédit.

Si votre formulaire utilise un groupe de validation spéficique, le prédicateur de type de champ continuera toujours à considérer *toutes* les contraintes de validation lorsqu'il essaie de deviner les types de champ (incluant les contraintes qui ne font pas partie du ou des groupes qui sont utilisés).

Prédiction des options de type de champ

En plus de pouvoir prédire le « type » d'un champ, Symfony peut aussi essayer de deviner les valeurs correctes d'un certain nombre d'options de champ.

Lorsque ces options sont définies, le champ sera affiché avec des attributs HTML spécifiques qui permettent la validation du champ côté client. Cependant, cela ne génère pas l'équivalent côté serveur (ex `Assert\Length`). Et puisque vous aurez besoin d'ajouter la validation côté serveur manuellement, ces options peuvent être déduites de cette information.

- `required` : L'option `required` peut être devinée grâce aux règles de validation (est-ce que le champ est `NotBlank` ou `NotNull`?) ou grâce aux métadonnées de Doctrine (est-ce que le champ est `nullable`?). Ceci est très utile car votre validation côté client va automatiquement correspondre à vos règles de validation.
- `max_length` : Si le champ est de type texte, alors l'option `max_length` peut être devinée grâce aux contraintes de validation (si `Length` ou `Range` sont utilisées) ou grâce aux métadonnées de Doctrine (via la longueur du champ).

Ces options de champ sont *uniquement* devinées si vous utilisez Symfony pour prédire le type des champs (c-a-d en omettant ou en passant `null` en tant que deuxième argument de la méthode `add()`).

Si vous voulez changer une des valeurs prédites, vous pouvez la surcharger en passant l'option au tableau des options de champ

Listing 12-19
```
->add('task', null, array('max_length' => 4))
```

Afficher un Formulaire dans un Template

Jusqu'içi, vous avez vu comment un formulaire entier peut être affiché avec seulement une ligne de code. Bien sûr, vous aurez probablement besoin de bien plus de flexibilité :

Listing 12-20
```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}
{{ form_start(form) }}
    {{ form_errors(form) }}

    {{ form_row(form.task) }}
    {{ form_row(form.dueDate) }}

    <input type="submit" />
{{ form_end(form) }}
```

Jetons un coup d'oeil à chaque partie :

- `form_start(form)` - Affiche la balise d'ouverture form.

- `form_errors(form)` - Affiche les erreurs globales du formulaire (les erreurs spécifiques à chaque champ seront affichées à côté des champs);
- `form_row(form.dueDate)` - Affiche le label, les erreurs éventuelles et le widget HTML d'un champ donné (ex `dueDate`) dans un élément `div` (par défaut);
- `form_end()` - Affiche la balise de fermeture du formulaire ainsi que tous les champs qui n'ont pas encore été affichés. C'est utile pour afficher les champs cachés et pour profiter de la *protection CSRF* automatique.

La plus grande partie du travail est effectuée par la fonction d'aide `form_row`, qui rend le label, les erreurs et le widget HTML de chaque champ dans une balise `div` par défaut. Dans la section *Habillage de Formulaire (« Theming »)*, vous apprendrez comment le rendu de `form_row` peut être personnalisé à différents niveaux.

Vous pouvez accéder à la donnée courante de votre formulaire via `form.vars.value`:

Listing 12-21
```
{{ form.vars.value.task }}
```

Afficher chaque Champ à la Main

La fonction `form_row` est géniale, car, grâce à elle, vous pouvez afficher très rapidement chaque champ de votre formulaire (et les balises utilisées pour ce champ peuvent être personnalisées aussi). Mais comme la vie n'est pas toujours simple, vous pouvez aussi afficher chaque champ entièrement à la main. Le résultat décrit ci-dessous revient au même que lorsque vous avez utilisé la fonction `form_row` :

Listing 12-22
```
{{ form_start(form) }}
    {{ form_errors(form) }}

    <div>
        {{ form_label(form.task) }}
        {{ form_errors(form.task) }}
        {{ form_widget(form.task) }}
    </div>

    <div>
        {{ form_label(form.dueDate) }}
        {{ form_errors(form.dueDate) }}
        {{ form_widget(form.dueDate) }}
    </div>

<input type="submit" />

{{ form_end(form) }}
```

Si le label autogénéré pour un champ n'est pas tout à fait correct, vous pouvez le spécifier explicitement :

Listing 12-23
```
{{ form_label(form.task, 'Task Description') }}
```

Certains types de champ ont des options d'affichage supplémentaires qui peuvent être passées au widget. Ces options sont documentées avec chaque type, mais l'option `attr` est commune à tous les type et vous permet de modifier les attributs d'un élément de formulaire. Ce qui suit ajouterait la classe `task_field` au champ texte affiché :

Listing 12-24
```
{{ form_widget(form.task, {'attr': {'class': 'task_field'}}) }}
```

Si vous avez besoin d'afficher' des champs du formulaire « à la main » alors vous pouvez utiliser individuellement les valeurs des champs comme `id`, `name` et `label`. Par exemple, pour obtenir l'`id` :

Listing 12-25
```
{{ form.task.vars.id }}
```

Pour obtenir la valeur utilisée pour l'attribut name du champ de formulaire, vous devez utiliser la valeur `full_name` :

Listing 12-26
```
{{ form.task.vars.full_name }}
```

Fonctions de Référence des Templates Twig

Si vous utilisez Twig, un *manuel de référence* complet des fonctions d'affichage de formulaire est mis à votre disposition. Lisez-le afin de tout connaître sur les fonctions d'aide disponibles ainsi que les options qui peuvent être utilisées pour chacune d'elles.

Changer les attributs Action et Method d'un formulaire

Jusqu'ici, la fonction `form_start()` a été utilisée pour afficher la balise d'ouverture du formulaire et nous avons supposé que chaque formulaire était soumis à la même URL avec une requête de type POST. Dans certains cas, vous aurez besoin de changer ces paramètres. Vous pouvez le faire de différentes manières. Si vous construisez votre formulaire dans un contrôleur, vous pouvez utiliser `setAction()` et `setMethod()`:

Listing 12-27
```
$form = $this->createFormBuilder($task)
    ->setAction($this->generateUrl('target_route'))
    ->setMethod('GET')
    ->add('task', 'text')
    ->add('dueDate', 'date')
    ->getForm();
```

Cet exemple suppose que vous avez créé une route appelée `target_route` qui pointe vers le contrôleur qui va traiter le formulaire.

Dans *Créer des Classes de Formulaire*, vous apprendrez comment déporter le code qui construit votre formulaire dans des classes séparées. Lorsque vous utilisez une classe de formulaire externe dans votre contrôleur, vous pouvez passer les attributs `action` et `method` comme options de votre formulaire:

Listing 12-28
```
$form = $this->createForm(new TaskType(), $task, array(
    'action' => $this->generateUrl('target_route'),
    'method' => 'GET',
));
```

Enfin, vous pouvez surcharger ces attributs dans un template en les passant aux fonctions `form()` ou `form_start()` :

Listing 12-29
```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}
{{ form(form, {'action': path('target_route'), 'method': 'GET'}) }}

{{ form_start(form, {'action': path('target_route'), 'method': 'GET'}) }}
```

Si la méthode n'est pas GET ou POST, mais PUT, PATCH ou DELETE, Symfony2 insèrera un nouveau champ caché avec le nom `_method` qui stockera cette méthode. Le formulaire sera soumis dans une requête POST classique mais le routeur de Symfony2 sera capable de détecter le paramètre `_method` et interprètera la requête comme une requête PUT, PATCH ou DELETE. Lisez le chapitre du Cookbook « *Comment utiliser des méthodes HTTP autres que GET et POST dans les routes* » pour obtenir plus d'informations à ce sujet.

Créer des Classes de Formulaire

Comme vous l'avez vu, un formulaire peut être créé et utilisé directement dans un contrôleur. Cependant, une meilleure pratique est de construire le formulaire dans une classe PHP séparée et autonome, qui peut ainsi être réutilisée n'importe où dans votre application. Créez une nouvelle classe qui va héberger la logique de construction du formulaire « task »:

Listing 12-30
```php
// src/Acme/TaskBundle/Form/Type/TaskType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('task');
        $builder->add('dueDate', null, array('widget' => 'single_text'));
    }

    public function getName()
    {
        return 'task';
    }
}
```

Cette nouvelle classe contient toutes les directives nécessaires à la création du formulaire « task » (notez que la méthode `getName()` doit retourner un identifiant unique pour ce « type » de formulaire). Il peut être utilisé pour construire rapidement un objet formulaire dans le contrôleur:

Listing 12-31
```php
// src/Acme/TaskBundle/Controller/DefaultController.php

// ajoutez cette nouvelle déclaration « use » en haut de la classe
use Acme\TaskBundle\Form\Type\TaskType;

public function newAction()
{
    $task = // ...
    $form = $this->createForm(new TaskType(), $task);

    // ...
}
```

Placer la logique du formulaire dans sa propre classe signifie que le formulaire peut être réutilisé facilement ailleurs dans votre projet. C'est la meilleure manière de créer des formulaires, mais le choix final vous revient.

Définir la `data_class`

Chaque formulaire a besoin de savoir le nom de la classe qui détient les données sous-jacentes (par exemple : `Acme\TaskBundle\Entity\Task`). Généralement, cette information est devinée grâce à l'objet passé en second argument de la méthode `createForm` (c-a-d `$task`). Plus tard, quand vous commencerez à imbriquer des formulaires les uns avec les autres, cela ne sera plus suffisant. Donc, bien que facultatif, c'est généralement une bonne idée de spécifier explicitement l'option `data_class` en ajoutant ce qui suit à votre classe formulaire:

Listing 12-32
```php
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Acme\TaskBundle\Entity\Task',
    ));
}
```

Lorsque vous associez un formulaire à un objet, tous les champs sont mappés. Chaque champ du formulaire qui n'existe pas dans l'objet associé entrainera la levée d'une exception.

Dans le cas où vous avez besoin de champs supplémentaires dans le formulaire (par exemple une checkbox « Acceptez-vous les conditions d'utilisation ») qui ne doit pas être mappée à l'objet sous-jacent, vous devez définir l'option `mapped` à `false`:

Listing 12-33
```php
use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add('task');
    $builder->add('dueDate', null, array('mapped' => false));
}
```

De plus, s'il y a des champs dans le formulaire qui ne sont pas inclus dans les données soumises, ces champs seront définis à `null`.

La donnée du champ est accessible dans un contrôleur de cette manière:

Listing 12-34
```php
$form->get('dueDate')->getData();
```

Définir vos formulaire en tant que services

Définir votre formulaire en tant que service est une bonne pratique, et le rend très facilement utilisable dans votre application.

Listing 12-35
```yaml
# src/Acme/TaskBundle/Resources/config/services.yml
services:
    acme_demo.form.type.task:
        class: Acme\TaskBundle\Form\Type\TaskType
        tags:
            - { name: form.type, alias: task }
```

Et c'est tout! Maintenant, vous pouvez utiliser votre type de formulaire directement dans un contrôleur:

Listing 12-36
```php
// src/Acme/TaskBundle/Controller/DefaultController.php
// ...

public function newAction()
{
    $task = ...;
    $form = $this->createForm('task', $task);

    // ...
}
```

ou même l'utiliser au sein d'un autre type de formulaire:

Listing 12-37
```php
// src/Acme/TaskBundle/Form/Type/ListType.php
// ...

class ListType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // ...

        $builder->add('someTask', 'task');
    }
}
```

Lisez *form-cookbook-form-field-service* pour plus d'informations.

Formulaires et Doctrine

Le but d'un formulaire est de traduire les données d'un objet (par exemple : `Task`) en un formulaire HTML et puis de transcrire en retour les données soumises par l'utilisateur à l'objet original. En tant que tel, le fait de persister l'objet `Task` dans la base de données n'a rien à voir avec les formulaires. Mais, si vous avez configuré la classe `Task` de telle sorte qu'elle soit persistée via Doctrine (c-a-d que vous avez ajouté des *métadonnées de correspondance*), alors sa persistance peut être effectuée après la soumission d'un formulaire lorsque ce dernier est valide:

Listing 12-38
```php
if ($form->isValid()) {
    $em = $this->getDoctrine()->getManager();
    $em->persist($task);
    $em->flush();

    return $this->redirect($this->generateUrl('task_success'));
}
```

Si, pour une quelconque raison, vous n'avez pas accès à votre objet `$task` d'origine, vous pouvez le récupérer depuis le formulaire:

Listing 12-39
```php
$task = $form->getData();
```

Pour plus d'informations, voir le *chapitre sur l'ORM Doctrine*.

Le plus important est de comprendre que lorsque le formulaire est lié (« bindé »), les données soumises sont transférées à l'objet sous-jacent immédiatement. Si vous souhaitez persister ces données, vous avez simplement besoin de persister l'objet lui-même (qui contient déjà les données soumises).

Formulaires imbriqués

Souvent, vous allez vouloir construire un formulaire qui inclura des champs de beaucoup d'objets différents. Par exemple, un formulaire de souscription pourrait contenir des données appartenant à un objet `User` ainsi qu'à plusieurs objets `Address`. Heureusement, gérer cela est facile et naturel avec le composant formulaire.

Imbriquer un objet unique

Supposez que chaque `Task` appartienne à un simple objet `Category`. Commencez, bien sûr, par créer l'objet `Category`:

Listing 12-40
```php
// src/Acme/TaskBundle/Entity/Category.php
namespace Acme\TaskBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Category
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

Ensuite, ajoutez une nouvelle propriété `category` à la classe `Task`:

Listing 12-41
```php
// ...

class Task
{
    // ...

    /**
     * @Assert\Type(type="Acme\TaskBundle\Entity\Category")
     */
    protected $category;

    // ...

    public function getCategory()
    {
        return $this->category;
    }

    public function setCategory(Category $category = null)
    {
        $this->category = $category;
    }
}
```

Maintenant que votre application a été mise à jour pour refléter nos nouvelles conditions requises, créez une classe formulaire afin que l'objet `Category` puisse être modifié par l'utilisateur:

Listing 12-42
```php
// src/Acme/TaskBundle/Form/Type/CategoryType.php
namespace Acme\TaskBundle\Form\Type;
```

```
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\OptionsResolver\OptionsResolverInterface;
7
8  class CategoryType extends AbstractType
9  {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder->add('name');
13     }
14
15     public function setDefaultOptions(OptionsResolverInterface $resolver)
16     {
17         $resolver->setDefaults(array(
18             'data_class' => 'Acme\TaskBundle\Entity\Category',
19         ));
20     }
21
22     public function getName()
23     {
24         return 'category';
25     }
26 }
```

Le but final est d'autoriser la `Category` d'une `Task` à être modifiée directement dans le formulaire de la tâche. Pour accomplir cela, ajoutez un champ `category` à l'objet `TaskType` dont le type est une instance de la nouvelle classe `CategoryType` :

Listing 12-43
```
1  use Symfony\Component\Form\FormBuilderInterface;
2
3  public function buildForm(FormBuilderInterface $builder, array $options)
4  {
5      // ...
6
7      $builder->add('category', new CategoryType());
8  }
```

Les champs de `CategoryType` peuvent maintenant être affichés à côté de ceux de la classe `TaskType`. Pour activer la validation sur CategoryType, ajoutez l'option `cascade_validation` à `TaskType`:

Listing 12-44
```
1  public function setDefaultOptions(OptionsResolverInterface $resolver)
2  {
3      $resolver->setDefaults(array(
4          'data_class' => 'Acme\TaskBundle\Entity\Task',
5          'cascade_validation' => true,
6      ));
7  }
```

Affichez les champs de `Category` de la même manière que les champs de la classe `Task` :

Listing 12-45
```
1  {# ... #}
2
3  <h3>Category</h3>
4  <div class="category">
5      {{ form_row(form.category.name) }}
```

```
</div>

{# ... #}
```

Lorsque l'utilisateur soumet le formulaire, les données soumises pour les champs de `Category` sont utilisées pour construire une instance de `Category` qui est ensuite affectée au champ `category` de l'instance de *Task*`.

L'instance `Category` est accessible naturellement via `$task->getCategory()` et peut être persistée dans la base de données ou utilisée de toute autre manière que vous souhaitez.

Imbriquer une Collection de Formulaires

Vous pouvez aussi imbriquer une collection de formulaires dans un formulaire (imaginez un formulaire `Catégorie` avec plusieurs sous-formulaires `Produit`). Cela peut être accompli en utilisant le type de champ `collection`.

Pour plus d'informations, lisez le chapitre du cookbook "*Comment imbriquer une Collection de Formulaires*" et le chapitre sur le type de champ *collection*.

Habillage de Formulaire (« Theming »)

Chaque partie de l'affichage d'un formulaire peut être personnalisée. Vous êtes libre de changer la manière dont chaque « partie » du formulaire est affichée, de changer les balises utilisées pour afficher les erreurs, ou même de personnaliser la manière dont la balise `textarea` doit être affichée. Tout est permis, et différentes personnalisations peuvent être utilisées à différents endroits.

Symfony utilise des templates pour afficher chaque partie d'un formulaire, comme les balises `tags`, les balises `input`, les messages d'erreur et tout le reste.

Dans Twig, chaque « fragment » de formulaire est représenté par un bloc Twig. Pour personnaliser n'importe quelle partie d'un formulaire, vous avez juste besoin de réécrire le bloc approprié.

En PHP, chaque « fragment » de formulaire est affiché via un fichier de template individuel. Pour personnaliser n'importe quelle partie d'un formulaire, vous avez juste besoin de réécrire le template existant en en créant un nouveau.

Pour comprendre comment tout cela fonctionne, personnalisons le fragment `form_row` et ajoutons l'attribut « class » à l'élément `div` qui entoure chaque ligne. Pour faire cela, créez un nouveau fichier de template qui va stocker la nouvelle balise :

Listing 12-46

```
{# src/Acme/TaskBundle/Resources/views/Form/fields.html.twig #}
{% block form_row %}
{% spaceless %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endspaceless %}
{% endblock form_row %}
```

Le fragment de formulaire `form_row` est utilisé pour afficher la plupart des champs via la fonction `form_row`. Pour dire au composant formulaire d'utiliser votre nouveau fragment `form_row` défini ci-dessus, ajoutez ce qui suit en haut du template qui affiche le formulaire :

Listing 12-47

```twig
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}
{% form_theme form 'AcmeTaskBundle:Form:fields.html.twig' %}

{% form_theme form 'AcmeTaskBundle:Form:fields.html.twig'
'AcmeTaskBundle:Form:fields2.html.twig' %}

{{ form(form) }}
```

La balise `form_theme` (dans Twig) « importe » les fragments définis dans le template donné et les utilise lorsqu'il affiche le formulaire. En d'autres termes, quand la fonction `form_row` est appelée plus tard dans ce template, elle va utiliser le bloc `form_row` de votre thème personnalisé (à la place du bloc par défaut `form_row` qui est fourni avec Symfony).

Votre thème personnalisé n'a pas besoin de surcharger tous les blocs. Lorsqu'il affiche un bloc qui n'est pas surchargé par votre thème personnalisé, le moteur de thème se rabattra sur le thème global (défini au niveau du bundle).

Si plusieurs thèmes personnalisés sont fournis, ils seront pris selon l'ordre dans lequel ils sont listés avant que le thème global ne soit pris en compte.

Pour personnaliser n'importe quelle portion d'un formulaire, vous devez juste réécrire le fragment approprié. Connaître exactement quel bloc ou fichier réécrire est le sujet de la prochaine section.

Listing 12-48
```twig
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

{% form_theme form with 'AcmeTaskBundle:Form:fields.html.twig' %}

{% form_theme form with ['AcmeTaskBundle:Form:fields.html.twig',
'AcmeTaskBundle:Form:fields2.html.twig'] %}
```

Pour plus de précisions, lisez *Comment personnaliser le rendu de formulaire*.

Nommage de Fragment de Formulaire

Dans Symfony, chaque partie d'un formulaire qui est affiché - éléments de formulaire HTML, erreurs, labels, etc - est définie dans un thème de base, qui est une collection de blocs dans Twig et une collection de fichiers de template dans PHP.

Dans Twig, chaque bloc nécessaire est défini dans un unique fichier de template (*form_div_layout.html.twig*[12]) qui se trouve dans le *Twig Bridge*[13]. Dans ce fichier, vous pouvez voir chaque bloc requis pour afficher un formulaire et chaque type de champ par défaut.

En PHP, les fragments sont des fichiers de template individuels. Par défaut, ils sont situés dans le répertoire *Resources/views/Form* du bundle du framework (*voir sur GitHub*[14]).

Chaque nom de fragment suit le même schéma de base et est divisé en deux parties, séparées par un unique underscore (_). Voici quelques exemples :

- `form_row` - utilisé par `form_row` pour afficher la plupart des champs ;
- `textarea_widget` - utilisé par `form_widget` pour afficher un champ de type `textarea` ;
- `form_errors` - utilisé par `form_errors` pour afficher les erreurs d'un champ.

Chaque fragment suit le même schéma de base : `type_part`. La partie `type` correspond au *type* du champ qui doit être affiché (par exemple : `textarea`, `checkbox`, `date`, etc) alors que la partie `part` correspond à

12. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
13. https://github.com/symfony/symfony/tree/2.2/src/Symfony/Bridge/Twig
14. https://github.com/symfony/symfony/tree/master/src/Symfony/Bundle/FrameworkBundle/Resources/views/Form

ce qui va être affiché (par exemple : `label`, `widget`, `errors`, etc). Par défaut, il y a 4 *parties* possibles d'un formulaire qui peuvent être affichées :

`label`	(par exemple : `form_label`)	affiche le label du champ
`widget`	(par exemple : `form_widget`)	affiche la représentation HTML du champ
`errors`	(par exemple : `form_errors`)	affiche les erreurs du champ
`row`	(par exemple : `form_row`)	affiche la ligne entière du champ (label, widget, et erreurs)

Il y a en fait 2 autres *parties*, `rows``et ``rest`, mais vous ne devriez que rarement, voire jamais, avoir besoin de les réécrire.

En connaissant le type du champ (par exemple : `textarea`) et quelle partie de ce dernier vous souhaitez personnaliser (par exemple : `widget`), vous pouvez construire le nom du fragment qui a besoin d'être réécrit (par exemple : `textarea_widget`).

Héritage de Fragment de Template

Dans certains cas, le fragment que vous voulez personnaliser sera absent. Par exemple, il n'y a pas de fragment `textarea_errors` dans les thèmes fournis par défaut par Symfony. Alors comment les erreurs des textareas sont-elles affichées ?

La réponse est : via le fragment `form_errors`. Quand Symfony affiche les erreurs d'un champ de type textarea, il recherche en premier un fragment `textarea_errors` avant de se replier sur le fragment de secours `form_errors`. Chaque type de champ a un type *parent* (le type parent de `textarea` est `field`), et Symfony l'utilise si le fragment de base n'existe pas.

Donc, afin de réécrire les erreurs pour les champs `textarea` *seulement*, copiez le fragment `form_errors`, renommez-le en `textarea_errors` et personnalisez-le. Pour réécrire le rendu d'erreur par défaut pour *tous* les champs, copiez et personnalisez le fragment `form_errors` directement.

Le type « parent » de chaque type de champ est disponible dans la *référence de type de formulaire* pour chaque type de champ.

Habillage global de Formulaire

Dans l'exemple ci-dessus, vous avez utilisé la fonction d'aide `form_theme` (dans Twig) pour « importer » les fragments personnalisés de formulaire à l'intérieur de ce formulaire uniquement. Vous pouvez aussi dire à Symfony d'importer les personnalisations de formulaire à travers votre projet entier.

Twig

Pour inclure automatiquement les blocs personnalisés du template `fields.html.twig` créés plus tôt dans *tous* les templates, modifiez votre fichier de configuration d'application :

Listing 12-49
```
1 # app/config/config.yml
2 twig:
```

```
    form:
        resources:
            - 'AcmeTaskBundle:Form:fields.html.twig'
    # ...
```

Tous les blocs se trouvant dans le template `fields.html.twig` sont maintenant utilisés pour l'affichage de tous vos formulaires.

Personnaliser l'affichage de formulaire dans un fichier unique avec Twig

Dans Twig, vous pouvez aussi personnaliser un bloc de formulaire directement à l'intérieur du template nécessitant une personnalisation :

Listing 12-50
```twig
{% extends '::base.html.twig' %}

{# importe « _self » en tant qu'habillage du formulaire #}
{% form_theme form _self %}

{# effectue la personnalisation du fragment de formulaire #}
{% block form_row %}
    {# personnalisez l'affichage de la ligne du champ #}
{% endblock form_row %}

{% block content %}
    {# ... #}

    {{ form_row(form.task) }}
{% endblock %}
```

La balise `{% form_theme form _self %}` permet aux blocs de formulaire d'être personnalisés directement à l'intérieur du template qui va utiliser ces personnalisations. Utilisez cette méthode pour faire des personnalisations de formulaire qui ne seront nécessaires que dans un unique template.

Cette fonctionnalité (`{% form_theme form _self %}`) ne marchera *que* si votre template en étend un autre. Si ce n'est pas le cas, vous devrez faire pointer `form_theme` vers un template séparé.

PHP

Pour inclure automatiquement les templates personnalisés du répertoire `Acme/TaskBundle/Resources/views/Form` créés plus tôt dans *tous* les templates, modifiez votre fichier de configuration d'application :

Listing 12-51
```yaml
# app/config/config.yml
framework:
    templating:
        form:
            resources:
                - 'AcmeTaskBundle:Form'
# ...
```

Tous les fragments à l'intérieur du répertoire `Acme/TaskBundle/Resources/views/Form` sont maintenant utilisés pour l'affichage de tous vos formulaires.

Protection CSRF

CSRF - ou *Cross-site request forgery*[15] - est une méthode grâce à laquelle un utilisateur malveillant tente de faire soumettre inconsciemment des données à un utilisateur légitime qui n'avait pas l'intention de les soumettre. Heureusement, les attaques CSRF peuvent être contrées en utilisant un jeton CSRF à l'intérieur de vos formulaires.

La bonne nouvelle est que, par défaut, Symfony prend en charge et valide automatiquement les jetons CSRF pour vous. Cela signifie que vous pouvez profiter de la protection CSRF sans n'avoir rien à faire. En fait, chaque formulaire dans ce chapitre a profité de la protection CSRF !

La protection CSRF fonctionne en ajoutant un champ caché dans votre formulaire - appelé `_token` par défaut - qui contient une valeur que seuls vous et votre utilisateur connaissez. Cela garantit que l'utilisateur - et lui seulement - soumet les informations données. Symfony valide automatiquement la présence et l'exactitude de ce jeton.

Le champ `_token` est un champ caché et sera affiché automatiquement si vous incluez la fonction `form_end()` dans votre template, qui garantit que tous les champs non affichés sont délivrés en sortie.

Le jeton CSRF peut être personnalisé pour chacun des formulaires. Par exemple:

Listing 12-52
```php
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class TaskType extends AbstractType
{
    // ...

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class'      => 'Acme\TaskBundle\Entity\Task',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // une clé unique pour aider à la génération du jeton secret
            'intention'       => 'task_item',
        ));
    }

    // ...
}
```

Pour désactiver la protection CSRF, définissez l'option `csrf_protection` à false. Les personnalisations peuvent aussi être effectuées globalement dans votre projet. Pour plus d'informations, voir la section de *référence de configuration de formulaire*.

L'option `intention` est optionnelle, mais améliore grandement la sécurité du jeton généré en le rendant différent pour chaque formulaire.

Utiliser un formulaire sans classe

Dans la plupart des cas, un formulaire est associé à un objet, et les champs du formulaire affichent et stockent leurs données dans les propriétés d'un objet. C'est exactement ce que vous avez vu jusqu'ici dans ce chapitre avec la classe *Task*.

15. http://fr.wikipedia.org/wiki/Cross-site_request_forgery

Mais parfois, vous voudrez juste utiliser un formulaire sans une classe, et obtenir un tableau des données soumises. C'est en fait très facile:

Listing 12-53
```php
// Assurez vous d'importer le namespace Request namespace en haut de la classe
use Symfony\Component\HttpFoundation\Request;
// ...

public function contactAction(Request $request)
{
    $defaultData = array('message' => 'Type your message here');
    $form = $this->createFormBuilder($defaultData)
        ->add('name', 'text')
        ->add('email', 'email')
        ->add('message', 'textarea')
        ->getForm();

    $form->handleRequest($request);

    if ($form->isValid()) {
        // Les données sont un tableau avec les clés "name", "email", et "message"
        $data = $form->getData();
    }

    // ... affiche le formulaire
}
```

Par défaut, en fait, un formulaire part du principe que vous voulez travailler avec un tableau de données plutôt qu'avec un objet. Il y a exactement deux façons de changer ce comportement et d'associer le formulaire avec un objet à la place :

1. Passez un objet lors de la création du formulaire (comme premier argument de `createFormBuilder` ou deuxième argument de `createForm`);
2. Définissez l'option `data_class` de votre formulaire.

Si vous ne faites *pas* l'un ou l'autre, alors le formulaire retournera les données dans un tableau. Dans cet exemple, puisque `$defaultData` n'est pas un objet (et que l'option `data_class` n'est pas définie), `$form->getData()` retournera finalement un tableau.

Vous pouvez également accéder directement aux valeurs POST (dans ce cas "name") par le biais de l'objet Request, comme ceci:

Listing 12-54
```php
$this->get('request')->request->get('name');
```

Notez cependant que, dans la plupart des cas, utiliser la méthode getData() est le meilleur choix, puisqu'elle retourne la donnée (souvent un objet) après qu'elle soit transformée par le framework.

Ajouter la Validation

La seule pièce manquante est la validation. D'habitude, quand vous appelez `$form->isValid()`, l'objet est validé en parcourant les contraintes que vous appliquez à sa classe. Si le formulaire est associé à un objet (c-a-d si vous utilisez l'option `data_class` ou si vous passez un object à votre formulaire), c'est l'approche que vous voudrez utiliser dans la plupart des cas. lisez *Validation* pour plus de détails.

Mais si le formulaire n'est pas associé à un objet et que vous voulez plutôt utiliser un simple tableau pour soumettre vos données, comment ajouter des contraintes aux données de votre formulaire ?

La réponse est de définir les contraintes vous-même, et de les attacher à chacun des champs. L'approche globale est un peu plus expliquée dans le *chapitre validation*, mais voici un petit exemple:

Listing 12-55
```php
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;

$builder
    ->add('firstName', 'text', array(
        'constraints' => new Length(array('min' => 3)),
    ))
    ->add('lastName', 'text', array(
        'constraints' => array(
            new NotBlank(),
            new Length(array('min' => 3)),
        ),
    ))
;
```

Si vous utilisez les groupes de validation, vous devrez soit référencer un groupe `Default` lorsque vous créez le formulaire, ou soit définir le bon groupe à la contrainte que vous ajoutez.

Listing 12-56
```php
new NotBlank(array('groups' => array('create', 'update'))
```

Réflexions finales

Vous connaissez maintenant tout sur les bases nécessaires à la construction de formulaires complexes et fonctionnels pour votre application. Quand vous construisez des formulaires, gardez à l'esprit que le but premier d'un formulaire est de transcrire les données d'un objet (`Task`) en un formulaire HTML afin que l'utilisateur puisse modifier ces données. Le second objectif d'un formulaire est de prendre les données soumises par l'utilisateur et de les réappliquer à l'objet.

Il y a beaucoup plus à apprendre à propos du puissant monde des formulaires, par exemple comment gérer les *uploads de fichier avec Doctrine* ou comment créer un formulaire où un nombre dynamique de sous-formulaires peut être ajouté (par exemple : une liste de choses à faire où vous pouvez continuer d'ajouter des champs via Javascript avant de soumettre le formulaire). Voyez le cookbook pour ces sujets. Aussi, assurez-vous de vous appuyer sur la *documentation de référence des types de champ*, qui inclut des exemples de comment utiliser chaque type de champ et leurs options.

En savoir plus grâce au Cookbook

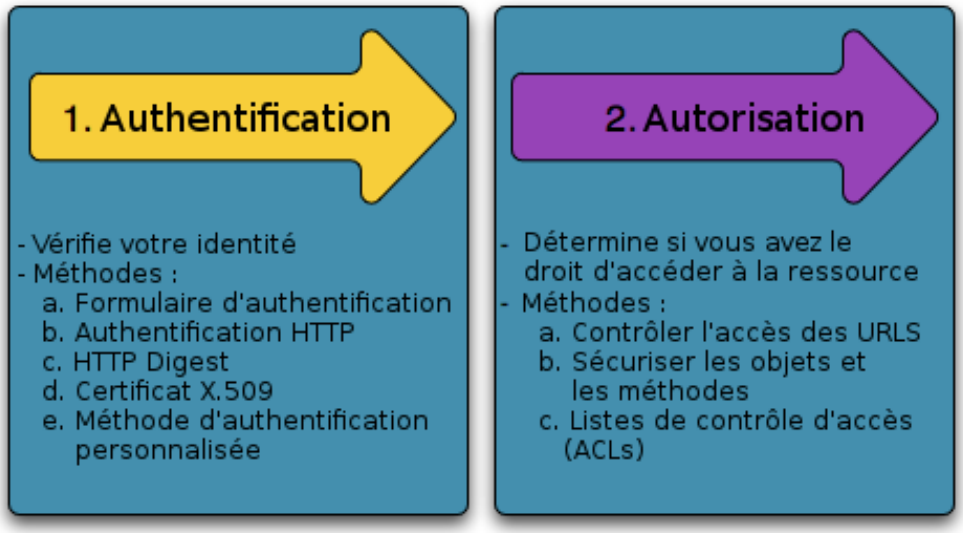
- *Comment gérer les uploads de fichier avec Doctrine*
- *Référence du Champ Fichier*
- *Créer des Types de Champ Personnalisés*
- *Comment personnaliser le rendu de formulaire*
- `/cookbook/form/dynamic_form_modification`
- *Comment utiliser les Convertisseurs de Données*

Chapter 13
La sécurité

La sécurité est un processus comprenant 2 étapes, dont le but est de prévenir un utilisateur d'accéder à une ressource à laquelle il n'a pas accès.

Dans la première étape du processus, le système de sécurité identifie l'utilisateur en lui demandant de soumettre une sorte d'identification. C'est ce qu'on appelle l'**authentification**, et cela signifie que le système cherche à savoir qui vous êtes.

Une fois que le système sait qui vous êtes, l'étape suivante est de déterminer si vous avez accès à une ressource donnée. Cette étape du processus est appelée **autorisation**, et cela signifie que le système vérifie si vous avez les privilèges pour exécuter certaines actions.



Comme la meilleure façon d'apprendre est par l'exemple, alors plongeons dans le vif du sujet.

Le *composant de sécurité* de Symfony est disponible en tant que bibliothèque indépendante, et peut être utilisé pour tout projet PHP.

Exemple simple: l'authentification HTTP

Le composant de sécurité peut être configuré grâce aux fichiers de configurations de l'application. En fait, la plupart des réglages de sécurité ne nécessitent que l'utilisation d'une configuration adéquate. La configuration suivante indique à Symfony de sécuriser toute URL correspondant au format `/admin/*` et de demander à l'utilisateur de s'authentifier en utilisant l'authentification HTTP (c'est-à-dire un bon vieux système avec login/mot de passe) :

Listing 13-1

```yaml
# app/config/config.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            http_basic:
                realm: "Secured Demo Area"

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }

    providers:
        in_memory:
             memory:
                users:
                    ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                    admin: { password: kitten, roles: 'ROLE_ADMIN' }

    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
```

La distribution Symfony Standard place la configuration de la sécurité dans un fichier séparé (`app/config/security.yml`). Si vous ne voulez pas utiliser un fichier séparé, vous pouvez mettre la configuration directement dans le fichier principal de configuration (`app/config/config.yml`).

Le résultat final de cette configuration est un système de sécurité entièrement fonctionnel, que l'on peut décrire de la manière suivante :

- Il y a 2 utilisateurs dans le système (`ryan` et `admin`);
- Les utilisateurs s'authentifient grâce à une authentification basique HTTP;
- Toute URL correspondant au format */admin/** est sécurisée, et seul l'utilisateur `admin` peut y accéder
- Toutes les URLs qui ne correspondent pas au format `/admin/*` sont accessibles par tous les utilisateurs (et l'utilisateur n'aura pas à s'authentifier).

Voyons rapidement comment la sécurité fonctionne et quel est le rôle de chaque élément de la configuration.

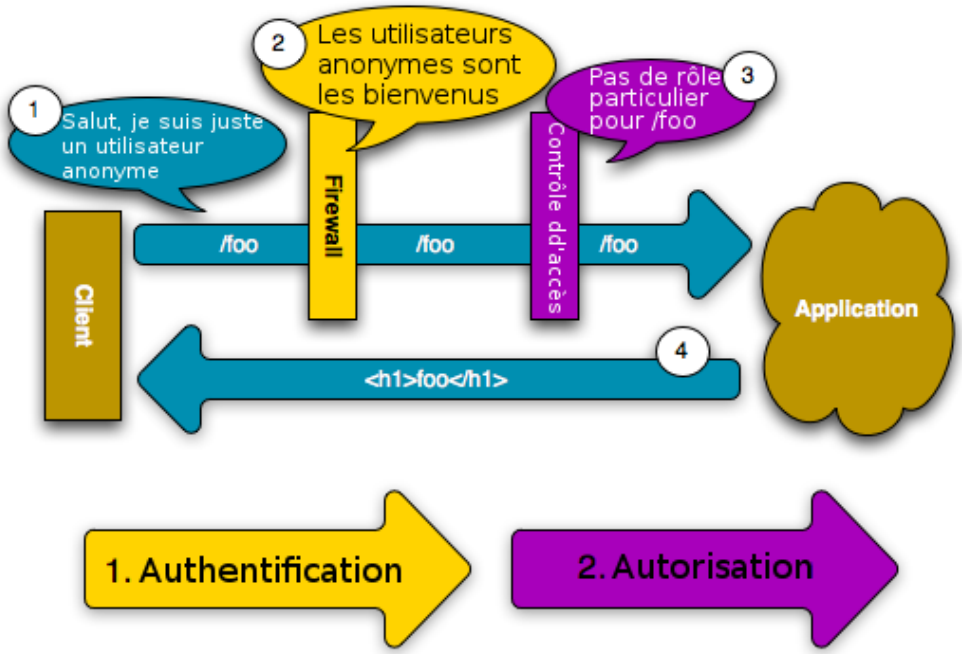
Comment fonctionne la sécurité : authentification et autorisation

Le système de sécurité de Symfony commence par déterminer qui est l'utilisateur (c'est l'authentification) puis il voit si l'utilisateur a accès à une ressource ou une URL.

Pare-feu (authentification)

Lorsqu'un utilisateur fait une requête à une URL qui est protégée par un pare-feu (firewall), le système de sécurité est activé. Le rôle du pare-feu est de déterminer si un utilisateur doit ou ne doit pas être authentifié, et s'il doit l'être, de retourner une réponse à l'utilisateur afin d'entamer le processus d'authentification.

Un pare-feu est activé lorsque l'URL d'une requête correspond à un `masque` d'expression régulière contenu dans la configuration du pare-feu. Dans cet exemple, le `masque` (`^/`) va correspondre à *toutes* les requêtes entrantes. Le fait que le pare-feu soit activé ne veut *pas* dire que la boite d'authentification HTTP contenant les champs « nom d'utilisateur » et « mot de passe » sera affichée pour chaque requête. Par exemple, tout utilisateur peut accéder `/foo` sans qu'on lui demande de s'authentifier.

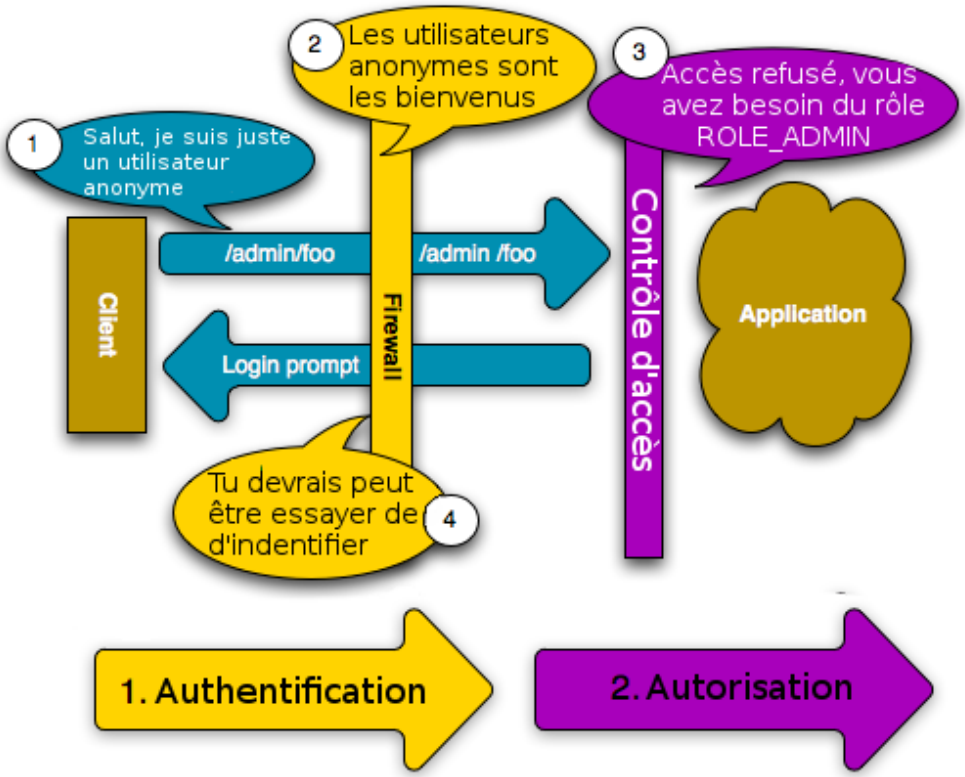


Cela fonctionne d'abord parce que le pare-feu autorise les *utilisateurs anonymes* grâce au paramètre de configuration `anonymous`. En d'autres termes, un pare-feu ne nécessite pas qu'un utilisateur soit totalement authentifié immédiatement. Et comme aucun `role` n'est nécessaire pour accéder l'URL `/foo``(dans la section ``access_control`), la requête peut être satisfaite sans jamais demander à l'utilisateur de s'authentifier.

Si vous supprimez la clé `anonymous`, le pare-feu va *toujours* demander à l'utilisateur de s'authentifier immédiatement.

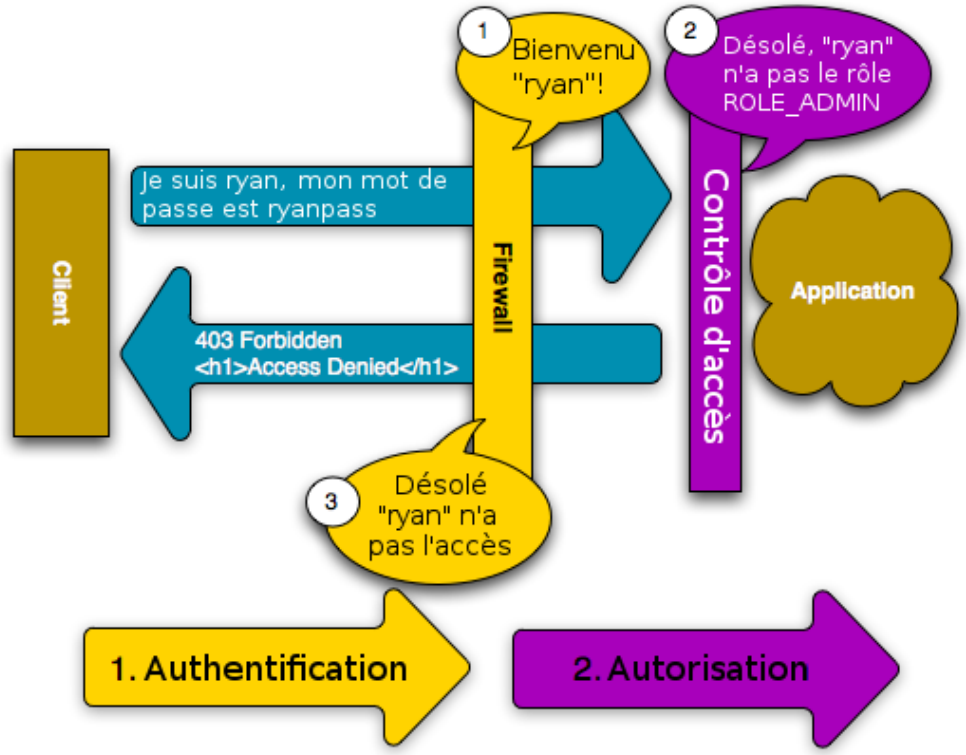
Contrôle d'accès (autorisation)

Par contre, si un utilisateur demande `/admin/foo`, le système se comporte différemment. C'est à cause de la section de la configuration `access_control` qui stipule que toute requête correspondant au masque d'expression régulière `^/admin` (c'est-à-dire `/admin` ou tout ce qui correspond à `/admin/*`) requiert le rôle `ROLE_ADMIN`. Les rôles sont à la base de la plupart des mécanismes d'autorisation : un utilisateur peut accéder à `/admin/foo` seulement s'il possède le role `ROLE_ADMIN`.



Comme précédemment, quand l'utilisateur fait une requête, le pare-feu ne lui demande pas de s'authentifier. Par contre, dès que la couche de contrôle d'accès refuse l'accès à l'utilisateur (parce que l'utilisateur anonyme ne possède pas le rôle `ROLE_ADMIN`), le pare-feu entre en action et initialise le processus d'authentification. Le processus d'authentification dépend du mécanisme d'authentification que vous utilisez. Par exemple, si vous utilisez la méthode d'authentification par formulaire de connexion, l'utilisateur sera redirigé à la page de formulaire de connexion. Si vous utilisez l'authentification HTTP, l'utilisateur recevra une réponse HTTP 401 et verra donc la boite contenant les champs login et mot de passe.

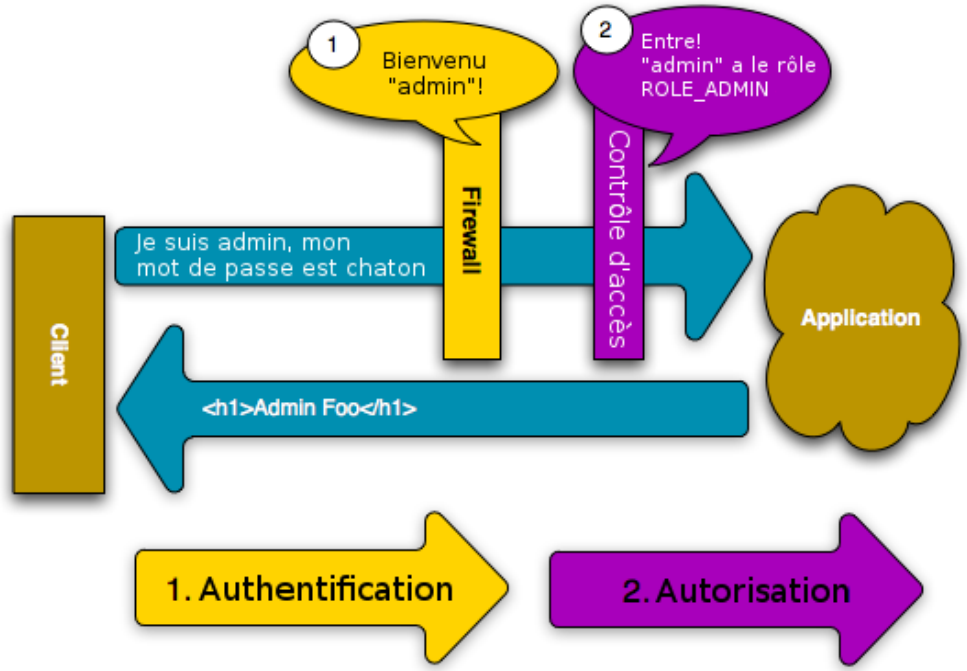
L'utilisateur a maintenant la possibilité de soumettre ses informations d'identification à l'application. Si ces informations sont valides, la requête initiale peut être lancée à nouveau.



Dans cet exemple, l'utilisateur `ryan``s'authentifie avec succès auprès du pare-feu. Mais comme ``ryan` n'a pas le rôle `ROLE_ADMIN`, il se verra refuser l'accès à `/admin/foo`. Enfin, cela veut dire que l'utilisateur verra un message indiquant que l'accès lui est refusé.

Quand Symfony refuse à l'utilisateur l'accès, l'utilisateur voit une page d'erreur et recevra un code d'erreur HTTP 403 (`Forbidden`). Vous pouvez personnaliser la page d'erreur pour refus d'accès en suivant les instructions se trouvant dans la page du cookbook *Pages d'erreurs<cookbook-error-pages-by-status-code>* pour personnaliser la page d'erreur 403.

Enfin, si l'utilisateur `admin` demande `/admin/foo`, un processus similaire se déroule, sauf que maintenant, après s'être authentifié, la couche de contrôle d'accès va laisser la requête s'exécuter :



Les étapes exécutées lorsqu'un utilisateur demande une ressource protégée sont simples, mais extrêmement flexibles. Comme vous le verrez plus tard, l'authentification peut être prise en charge de multiples façons, incluant les formulaires de connexion, les certificats X.509, ou les authentifications via Twitter. Quel que soit la méthode d'authentification, les étapes sont toujours les mêmes :

1. Un utilisateur accède à une ressource protégée;
2. L'application redirige l'utilisateur au formulaire de connexion;
3. L'utilisateur soumet ses informations d'identification (par exemple login/mot de passe);
4. Le pare-feu authentifie l'utilisateur;
5. L'utilisateur authentifié renvoie la requête initiale.

Le processus *exact* dépend en fait légèrement du mécanisme d'authentification que vous utilisez. Par exemple, lorsque le formulaire de connexion est utilisé, l'utilisateur soumet ses informations d'identification à une URL qui traite le formulaire (par exemple `/login_check`) et est ensuite redirigé à l'URL qu'il a demandée initialement (par exemple `/admin/foo`). Par contre, avec l'authentification HTTP, l'utilisateur soumet ses informations d'identification directement à l'URL initiale (par exemple `/admin/foo`) et la page est retournée dans la même requête (donc pas de redirection).

Ces comportements différents (types d'idiosyncrasie) ne devraient pas vous causer de problèmes, mais il est bon de les garder à l'esprit.

Vous apprendrez plus tard comment *tout* peut être sécurisé avec Symfony2, incluant certains contrôleurs, objets, ou même méthodes PHP.

Utilisation d'un formulaire de connexion traditionnel

Dans cette section, vous allez apprendre comment créer un formulaire de connexion basique qui continue d'utiliser les utilisateurs codés en dur que vous avez défini dans le fichier `security.yml`.

Pour charger les utilisateurs de la base de données, lisez *Comment charger les utilisateurs depuis la base de données (le fournisseur d'Entité)*. En lisant cet article et cette section, vous pouvez créer un système de connexion complet qui charge les utilisateurs dans la base de données.

Pour l'instant, vous avez vu comment protéger votre application derrière un pare-feu et ensuite comment protéger l'accès à certaines zones en utilisant les rôles. En utilisant l'authentification HTTP, vous pouvez sans effort profiter de la boite login/mot de passe offert par tous les navigateurs. Mais Symfony comprend plusieurs mécanismes d'authentification par défaut. Pour plus de détails sur chacun d'eux, référez-vous à la documentation de *référence sur la configuration de la sécurité*.

Dans cette section, vous allez améliorer le processus en autorisant l'utilisateur à s'authentifier via un formulaire de connexion traditionnel.

D'abord, activez le formulaire de connexion (« form login ») de votre pare-feu:

Listing 13-2
```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            form_login:
                login_path:  /login
                check_path:  /login_check
```

Si vous ne voulez pas personnaliser les valeurs de `login_path` ou `check_path` (les valeurs utilisées ici sont celles par défaut), vous pouvez raccourcir votre configuration :

Listing 13-3
```yaml
form_login: ~
```

Maintenant, quand le système de sécurité initie le processus d'authentification, il va rediriger l'utilisateur au formulaire de connexion (`/login` by default). L'implémentation de ce formulaire de connexion est de toute évidence votre responsabilité. Tout d'abord, créez 2 routes : une qui affiche le formulaire de connexion (ici, `/login`) et une qui va prendre en charge la soumission du formulaire (ici, `/login_check`) :

Listing 13-4
```yaml
# app/config/routing.yml
login:
    pattern:   /login
    defaults:  { _controller: AcmeSecurityBundle:Security:login }
login_check:
    pattern:   /login_check
```

Vous *n'avez pas* à implémenter un contrôleur pour l'URL `/login_check` car le pare-feu va automatiquement intercepter et traiter tout formulaire soumis à cette URL.

New in version 2.1: Dans Symfony 2.1, vous *devez* avoir des routes configurées pour vos URLs `login_path` (ex `/login`), `check_path` (ex `/login_check`) et `logout` (ex `/logout` - voir Se déconnecter).

Veuillez noter que le nom de la route `login` n'est pas important. Ce qui importe est que l'URL de la route (`login`) corresponde à la valeur de `login_path`, car c'est là que le système de sécurité va rediriger les utilisateurs qui doivent se connecter.

Ensuite, créez un contrôleur qui va afficher le formulaire de connexion:

Listing 13-5
```php
// src/Acme/SecurityBundle/Controller/SecurityController.php;
namespace Acme\SecurityBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;

class SecurityController extends Controller
{
    public function loginAction()
    {
        $request = $this->getRequest();
        $session = $request->getSession();
        // get the login error if there is one
        if ($request->attributes->has(SecurityContext::AUTHENTICATION_ERROR)) {
            $error = $request->attributes->get(SecurityContext::AUTHENTICATION_ERROR);
        } else {
            $error = $session->get(SecurityContext::AUTHENTICATION_ERROR);
            $session->remove(SecurityContext::AUTHENTICATION_ERROR);
        }
        return $this->render('AcmeSecurityBundle:Security:login.html.twig', array(
            // last username entered by the user
            'last_username' => $session->get(SecurityContext::LAST_USERNAME),
            'error'         => $error,
        ));
    }
}
```

Ne vous laissez pas impressionner par le contrôleur. Comme vous allez le voir dans un moment, lorsque l'utilisateur soumet le formulaire, le système de sécurité prend en charge automatiquement le formulaire soumis. Si l'utilisateur venait à soumettre un login ou un mot de passe invalide, ce formulaire lit les erreurs de soumission du système de sécurité afin qu'elles soient ensuite affichées à l'utilisateur.

En d'autres termes, votre rôle est d'afficher le formulaire de connexion et toute erreur qui aurait pu survenir, mais c'est le système de sécurité lui-même qui prend en charge la validation du login et du mot de passe et qui authentifie l'utilisateur.

Il ne nous reste qu'à créer le template correspondant :

Listing 13-6
```twig
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Login :</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Mot de passe :</label>
```

```
    <input type="password" id="password" name="_password" />

    {#
        Si vous voulez contrôler l'URL vers laquelle l'utilisateur est redirigé en cas de
succès
        (plus de détails ci-dessous)
        <input type="hidden" name="_target_path" value="/account" />
    #}

    <button type="submit">login</button>
</form>
```

La variable `error` passée au template est une instance de *`AuthenticationException`*[1]. Elle peut contenir plus d'informations - et même des informations sensibles - à propos de l'échec de l'authentification, alors utilisez là judicieusement !

Le formulaire a très peu d'exigence. D'abord, en soumettant le formulaire à `/login_check` (via la route `login_check`), le système de sécurité va intercepter la soumission du formulaire et traiter le formulaire automatiquement. Ensuite, le système de sécurité s'attend à ce que les champs soumis soient nommés `_username` et `_password` (le nom de ces champs peut être *configuré*).

Et c'est tout ! Lorsque vous soumettez le formulaire, le système de sécurité va automatiquement vérifier son identité et va soit authentifier l'utilisateur, soit renvoyer l'utilisateur au formulaire de connexion, où les erreurs vont être affichées.

Récapitulons tout le processus :

1. L'utilisateur cherche à accéder une ressource qui est protégée;
2. Le pare-feu initie le processus d'authentification en redirigeant l'utilisateur au formulaire de connexion (`/login`);
3. La page `/login` affiche le formulaire de connexion en utilisant la route et le formulaire créés dans cet exemple.
4. L'utilisateur soumet le formulaire de connexion à `/login_check`;
5. Le système de sécurité intercepte la requête, vérifie les informations d'identification soumises par l'utilisateur, authentifie l'utilisateur si elles sont correctes et renvoie l'utilisateur au formulaire de connexion si elles ne le sont pas.

Par défaut, si les informations d'identification sont correctes, l'utilisateur va être redirigé à la page originale qu'il avait demandée (par exemple `/admin/foo`). Si l'utilisateur est allé directement au formulaire de connexion, il sera redirigé à la page d'accueil. Cela peut être entièrement configuré, en vous permettant, par exemple, de rediriger l'utilisateur vers une URL spécifique.

Pour plus de détails, et savoir comment personnaliser le processus de connexion par formulaire en général, veuillez vous reporter à *Comment personnaliser votre formulaire de login*.

1. http://api.symfony.com/master/Symfony/Component/Security/Core/Exception/AuthenticationException.html

Éviter les erreurs courantes

Lorsque vous configurez le formulaire de connexion, faites attention aux pièges.

1. Créez les routes adéquates

D'abord, assurez-vous que vous avez défini les routes `/login` et `/login_check` correctement et qu'elles correspondent aux valeurs de configuration `login_path` et `check_path`. Une mauvaise configuration ici pourrait vouloir dire que vous seriez redirigé à une page 404 au lieu de la page de connexion, ou que la soumission du formulaire ne fasse rien (vous ne verriez que le formulaire de connexion encore et encore).

2. Assurez-vous que la page de connexion n'est pas sécurisée

Aussi, assurez-vous que la page de connexion ne requiert *pas* un rôle particulier afin d'être affichée. Par exemple, la configuration suivante - qui nécessite le rôle `ROLE_ADMIN` pour toutes les URLs (incluant l'URL `/login`), va provoquer une boucle de redirection :

Listing 13-7
```
access_control:

    - { path: ^/, roles: ROLE_ADMIN }
```

Il suffit d'enlever le contrôle d'accès pour l'URL `/login` URL pour corriger le problème :

Listing 13-8
```
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/, roles: ROLE_ADMIN }
```

Aussi, si votre pare-feu n'autorise *pas* les utilisateurs anonymes, vous devrez créer un pare-feu spécial qui permet l'accès à l'utilisateur anonyme d'accéder la page de connexion :

Listing 13-9
```
firewalls:
    login_firewall:
        pattern:    ^/login$
        anonymous:  ~
    secured_area:
        pattern:    ^/
        form_login: ~
```

3. Assurez-vous que ``/login_check`` est derrière un pare-feu

Ensuite, assurez-vous que l'URL `check_path` (ici, `/login_check`) est derrière le pare-feu que vous utilisez pour le formulaire de connexion (dans cet exemple, le pare-feu unique correspond à *toutes* les URLs, incluant `/login_check`). Si `/login_check` n'est pris en charge par aucun pare-feu, vous obtiendrez une exception `Unable to find the controller for path "/login_check"`.

4. Plusieurs pare-feu ne partagent pas de contexte de sécurité

Si vous utilisez plusieurs pare-feu et que vous vous authentifiez auprès d'un pare-feu, vous ne serez *pas* automatiquement authentifié auprès des autres pare-feu automatiquement. Différents pare-feu sont comme plusieurs systèmes de sécurité. C'est pourquoi, pour la plupart des applications, avoir un seul pare-feu est suffisant.

Autorisation

La première étape en sécurité est toujours l'authentification : le processus de vérifier l'identité de l'utilisateur. Avec Symfony, l'authentification peut être faite de toutes les façons voulues - au travers d'un formulaire de connexion, de l'authentification HTTP, ou même de facebook.

Une fois l'utilisateur authentifié, l'autorisation commence. L'autorisation fournit une façon standard et puissante de décider si un utilisateur peut accéder une ressource (une URL, un objet du modèle, un appel de méthode...). Cela fonctionne en assignant des rôles à chaque utilisateur, et d'ensuite en requérant différents rôles pour différentes ressources.

Le processus d'autorisation comporte 2 aspects :

1. Un utilisateur possède un ensemble de rôles;
2. Une ressource requiert un rôle spécifique pour être atteinte.

Dans cette section, vous verrez en détail comment sécuriser différentes ressources (ex. URLs, appels de méthodes...) grâce aux rôles. Plus tard, vous apprendrez comment les rôles peuvent être créés et assignés aux utilisateurs.

Sécurisation d'URLs spécifiques

La façon la plus simple pour sécuriser une partie de votre application est de sécuriser un masque d'URL au complet. Vous avez déjà vu dans le premier exemple de ce chapitre, où tout ce qui correspondait à l'expression régulière `^/admin` nécessite le role `ROLE_ADMIN`.

Vous pouvez définir autant de masque d'URL que vous voulez - chacune étant une expression régulière.

Listing 13-10

```yaml
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/admin/users, roles: ROLE_SUPER_ADMIN }
        - { path: ^/admin, roles: ROLE_ADMIN }
```

En préfixant votre chemin par `^`, vous vous assurez que seules les URLs *commençant* par le masque correspondent. Par exemple, un chemin spécifiant simplement `/admin` (sans le `^`) reconnaîtra une url du type `/admin/foo` mais aussi `/foo/admin`.

Pour chaque requête entrante, Symfony essaie de trouver une règle d'accès de contrôle (la première gagne). Si l'utilisateur n'est pas encore authentifié, le processus d'authentification est initié (c'est-à-dire que l'utilisateur a une chance de se connecter). Mais si l'utilisateur *est* authentifié, mais qu'il ne possède pas le rôle nécessaire, une exception *`AccessDeniedException`*[2] est lancée, qui peut être attrapée et convertie en une belle page d'erreur « accès refusé » présentée à l'utilisateur. Voir *Comment personnaliser les pages d'erreur* pour plus d'informations.

Comme Symfony utilise la première règle d'accès de contrôle qui correspond, une URL comme `/admin/users/new` correspondra à la première règle et ne nécessitera que le rôle `ROLE_SUPER_ADMIN`. Tout URL comme `/admin/blog` correspondra à la seconde règle et nécessitera donc `ROLE_ADMIN`.

Sécuriser par IP

Dans certaines situations qui peuvent survenir, vous aurez besoin de restreindre l'accès à une route donnée basée sur une IP. C'est particulièrement le cas des *Edge Side Includes* (ESI), par exemple, qui utilisent une route nommée « _internal ». Lorsque les ESI sont utilisés, la route _internal est requise par la passerelle de cache pour activer différentes options de cache pour les portions d'une même page. Dans la Standard Edition, cette route est préfixée par défaut par ^/_internal (en supposant que vous avez décommenté ces lignes dans le fichier de routage)

Ci-dessous un exemple de comment sécuriser une route d'un accès externe :

Listing 13-11

2. http://api.symfony.com/master/Symfony/Component/Security/Core/Exception/AccessDeniedException.html

```yaml
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/cart/checkout, roles: IS_AUTHENTICATED_ANONYMOUSLY, ip: 127.0.0.1 }
```

Sécuriser par canal

Tout comme la sécurisation basée sur IP, obliger l'usage d'SSL est aussi simple qu'ajouter une nouvelle entrée access_control :

Listing 13-12
```yaml
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/_internal, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel: https
}
```

Sécuriser un contrôleur

Protéger votre application en utilisant des masques d'URL est facile, mais pourrait ne pas offrir une granularité suffisante dans certains cas. Si nécessaire, vous pouvez facilement forcer l'autorisation dans un contrôleur:

Listing 13-13
```php
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...
public function helloAction($name)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }
    // ...
}
```

Vous pouvez aussi choisir d'installer et d'utiliser le Bundle `JMSSecurityExtraBundle`, qui peut sécuriser un contrôleur en utilisant les annotations:

Listing 13-14
```php
use JMS\SecurityExtraBundle\Annotation\Secure;
/**
 * @Secure(roles="ROLE_ADMIN")
 */
public function helloAction($name)
{
    // ...
}
```

Pour plus d'informations, voir la documentation de *JMSSecurityExtraBundle*[3]. Si vous utilisez la distribution standard de Symfony, ce bundle est disponible par défaut. Sinon, vous pouvez facilement le télécharger et l'installer.

3. http://jmsyst.com/bundles/JMSSecurityExtraBundle/1.2

Sécuriser d'autres services

En fait, tout dans Symfony peut être protégé en utilisant une stratégie semblable à celle décrite dans les sections précédentes. Par exemple, supposez que vous avez un service (une classe PHP par exemple) dont la responsabilité est d'envoyer des courriels d'un utilisateur à un autre. Vous pouvez restreindre l'utilisation de cette classe - peu importe d'où vous l'utilisez - à des utilisateurs qui ont des rôles spécifiques.

Pour plus d'informations sur la manière d'utiliser le composant de sécurité pour sécuriser différents services et méthodes de votre application, voir *Comment sécuriser n'importe quel service ou méthode de votre application*.

Listes de contrôle d'accès (ACL): sécuriser des objets de la base de données

Imaginez que vous êtes en train de concevoir un système de blog où les utilisateurs peuvent écrire des commentaires sur les articles. Mais vous voulez qu'un utilisateur puisse éditer ses propres commentaires, mais pas les autres utilisateurs. Aussi, vous, en tant qu'administrateur, voulez pouvoir éditer *tous* les commentaires.

Le composant de sécurité comprend un système de liste de contrôle d'accès (Access Control List, ou ACL) que vous pouvez utiliser pour contrôler l'accès à des instances individuelles de votre système. *Sans* la liste d'accès de contrôle, vous pouvez sécuriser votre système pour que seulement certains utilisateurs puissent éditer les commentaires en général. Mais *avec* la liste d'accès de contrôle, vous pouvez restreindre ou autoriser l'accès à un commentaire en particulier.

Pour plus d'informations, reportez-vous à l'article du cookbook *Comment utiliser les Access Control Lists (ACLs) (« liste de contrôle d'accès » en français)*.

Les utilisateurs

Dans les sections précédentes, vous avez appris comment vous pouvez protéger différentes ressources en exigeant un ensemble de rôles pour une ressource. Cette section aborde l'autre aspect de l'autorisation : les utilisateurs.

D'où viennent les utilisateurs (*Fournisseurs d'utilisateurs*)

Au cours de l'authentification, l'utilisateur soumet ses informations d'identité (généralement un login et un mot de passe). La responsabilité du système d'authentification est de faire correspondre cette identité avec un ensemble d'utilisateurs. Mais d'où cet ensemble provient-il?

Dans Symfony2, les utilisateurs peuvent provenir de n'importe où - un fichier de configuration, une table de base de données, un service Web, ou tout ce que vous pouvez imaginer d'autre. Tout ce qui fournit un ou plusieurs utilisateurs au système d'authentification est appelé « fournisseur d'utilisateurs » (User Provider). Symfony2 comprend en standard deux des fournisseurs les plus utilisés : un qui charge ses utilisateurs depuis un fichier de configuration, et un autre qui charge ses utilisateurs d'une table de base de données.

Spécifier les utilisateurs dans un fichier de configuration

La manière la plus simple de définir des utilisateurs est de la faire directement dans un fichier de configuration. En fait, vous avez déjà vu cet exemple dans ce chapitre.

Listing 13-15
```yaml
# app/config/security.yml
security:
    # ...
    providers:
```

```
        default_provider:
            users:
                ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                admin: { password: kitten, roles: 'ROLE_ADMIN' }
```

Ce fournisseur d'utilisateurs est appelé fournisseur d'utilisateurs en mémoire (« in-memory ») car les utilisateurs ne sont pas sauvegardés dans une base de données. L'objet User est fourni par Symfony (*User*[4]).

Tout fournisseur d'utilisateur peut charger des utilisateurs directement de la configuration en spécifiant le paramètre de configuration `users` et en listant les utilisateurs en dessous.

Si votre login est complètement numérique (par exemple `77`) ou contient un tiret (par exemple `user-name`), vous devez utiliser une syntaxe alternative pour définir les utilisateurs en YAML:

Listing 13-16
```
users:
    - { name: 77, password: pass, roles: 'ROLE_USER' }
    - { name: user-name, password: pass, roles: 'ROLE_USER' }
```

Pour les petits sites, cette méthode est rapide et facile à mettre en place. Pour des systèmes plus complexes, vous allez vouloir charger vos utilisateurs de la base de données.

Charger les utilisateurs de la base de données

Si vous voulez charger vos utilisateurs depuis l'ORM Doctrine, vous pouvez facilement le faire en créant une classe `User``et en configurant le fournisseur d'entités (``entity` provider).

Un bundle de très grande qualité est disponible, qui permet de sauvegarder vos utilisateurs depuis l'ORM ou l'ODM de Doctrine. Apprenez-en plus sur le *FOSUserBundle*[5] sur GitHub.

Avec cette approche, vous devez d'abord créer votre propre classe `User`, qui va être sauvegardée dans la base de données.

Listing 13-17
```
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;
use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity
 */
class User implements UserInterface
{
    /**
     * @ORM\Column(type="string", length=255)
     */
    protected $username;
```

4. http://api.symfony.com/master/Symfony/Component/Security/Core/User/User.html
5. https://github.com/FriendsOfSymfony/FOSUserBundle

```
14      // ...
15  }
```

Pour ce qui concerne le système de sécurité, la seule exigence est que la classe User implémente l'interface *`UserInterface`*[6]. Cela signifie que le concept d'« utilisateur » peut être n'importe quoi, pour peu qu'il implémente cette interface.

New in version 2.1: Dans Symfony 2.1, la méthode `equals` a été retirée de la `UserInterface`. Si vous avez besoin de surcharger l'implémentation par défaut de la logique de comparaison, implémentez la nouvelle interface *`EquatableInterface`*[7].

L'objet User sera sérialisé et sauvegardé dans la session lors des requêtes, il est donc recommandé d'*implémenter l'interface Serializable*[8] dans votre classe User. Cela est spécialement important si votre classe `User` a une classe parente avec des propriétés privées.

Ensuite, il faut configurer le fournisseur d'utilisateur `entity` (`entity` user provider), le pointer vers la classe `User` :

Listing 13-18
```yaml
# app/config/security.yml
security:
    providers:
        main:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

Avec l'introduction de ce nouveau fournisseur, le système d'authentification va tenter de charger un objet `User` depuis la base de données en utilisant le champ `username` de cette classe.

Cet exemple ne vous montre que les principes de base du fournisseur `entity`. Pour un exemple complet et fonctionnel, veuillez lire *Comment charger les utilisateurs depuis la base de données (le fournisseur d'Entité)*.

Pour en apprendre plus sur comment créer votre propre fournisseur (par exemple si vous devez charger des utilisateurs depuis un service Web), reportez-vous à *Comment créer un Fournisseur d'Utilisateur personnalisé*.

Encoder les mots de passe

Jusqu'à maintenant, afin de garder ça simple, les mots de passe des utilisateurs ont tous été conservés au format texte (qu'ils soient sauvegardés dans un fichier de configuration ou dans la base de données). Il est clair que dans une vraie application, vous allez vouloir encoder les mots de passe de vos utilisateurs pour des raisons de sécurité. Ceci est facile à accomplir en mappant votre classe User avec un des nombreux « encodeurs » intégrés.

Par exemple, pour rendre indéchiffrables les mots de passe de vos utilisateurs en utilisant `sha1`, suivez les instructions suivantes :

Listing 13-19

6. http://api.symfony.com/master/Symfony/Component/Security/Core/User/UserInterface.html
7. http://api.symfony.com/master/Symfony/Component/Security/Core/User/EquatableInterface.html
8. http://php.net/manual/en/class.serializable.php

```yaml
# app/config/security.yml
security:
    # ...
    providers:
        in_memory:
            users:
                ryan:  { password: bb87a29949f3a1ee0559f8a57357487151281386, roles:
'ROLE_USER' }
                admin: { password: 74913f5cd5f61ec0bcfdb775414c2fb3d161b620, roles:
'ROLE_ADMIN' }

    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm:   sha1
            iterations: 1
            encode_as_base64: false
```

En spécifiant les `itérations` à `1``et le paramètre ``encode_as_base64` à false, le mot de passe est simplement encrypté en utilisant l'algorithme `sha1``une fois, et sans aucun encodage additionnel. Vous pouvez maintenant calculer le mot de passe soit programmatiquement (c'est-à-dire ``hash('sha1', 'ryanpass')`) ou soit avec des outils en ligne comme *functions-online.com*[9]

Si vous créez vos utilisateurs dynamiquement (et que vous les sauvegardez dans une base de données), vous pouvez rendre l'algorithme de hachage plus complexe puis utiliser un objet d'encodage de mot de passe pour vous aider à encoder les mots de passe. Par exemple, supposez que votre objet User est un `Acme\UserBundle\Entity\User` (comme dans l'exemple ci-dessus). D'abord, configurez l'encodeur pour cet utilisateur :

Listing 13-20
```yaml
# app/config/security.yml
security:
    # ...

    encoders:
        Acme\UserBundle\Entity\User: sha512
```

Dans cet exemple, nous utilisons L'algorithme plus puissant `sha512`. Aussi, comme nous avons uniquement spécifié l'algorithme (`sha512`) sous forme de chaîne de caractères, le système va par défaut hacher votre mot de passe 5000 fois de suite et ensuite l'encoder en base64. En d'autres termes, le mot de passe a été très fortement obscurci pour ne pas qu'il puisse être décodé (c'est-à-dire que vous ne pouvez pas retrouver le mot de passe depuis le mot de passe haché).

New in version 2.2: Depuis Symfony 2.2 vous pouvez également utiliser l'encodeur de mot de passe PBKDF2.

Si vous avez une sorte de formulaire d'enregistrement pour les utilisateurs, vous devez pouvoir générer un mot de passe haché pour pouvoir le sauvegarder. Peu importe l'algorithme que vous avez configuré pour votre objet User, le mot de passe haché peut toujours être déterminé de la manière suivante depuis un contrôleur :

Listing 13-21
```php
$factory = $this->get('security.encoder_factory');
$user = new Acme\UserBundle\Entity\User();
```

9. http://www.functions-online.com/sha1.html

```
3
4  $encoder = $factory->getEncoder($user);
5  $password = $encoder->encodePassword('ryanpass', $user->getSalt());
6  $user->setPassword($password);
```

Récupérer l'objet User

Après l'authentification, l'objet `User` correspondant à l'utilisateur courant peut être récupéré via le service `security.context`. Depuis un contrôleur, cela ressemble à ça:

Listing 13-22
```
1  public function indexAction()
2  {
3      $user = $this->get('security.context')->getToken()->getUser();
4  }
```

Dans un contrôleur, vous pouvez utiliser le raccourci suivant :

Listing 13-23
```
1  public function indexAction()
2  {
3      $user = $this->getUser();
4  }
```

Les utilisateurs anonymes sont techniquement authentifiés, ce qui veut dire que la méthode `isAuthenticated()` sur un objet d'utilisateur anonyme va retourner true. Pour vérifier si un utilisateur est vraiment authentifié, vérifiez si l'utilisateur a le rôle `IS_AUTHENTICATED_FULLY`.

Dans un template Twig, cet objet est accessible via la clé `app.user`, qui appelle la méthode *`GlobalVariables::getUser()`*[10] :

Listing 13-24
```
1  <p>Username: {{ app.user.username }}</p>
```

Utiliser plusieurs fournisseurs d'utilisateurs

Chaque mécanisme d'authentification (par exemple authentification HTTP, formulaire de connexion, etc...) utilise exactement un fournisseur d'utilisateur (user provider), et va utiliser par défaut le premier fournisseur d'utilisateurs déclaré. Mais que faire si vous voulez déclarer quelques utilisateurs via la configuration et le reste des utilisateurs dans la base de données? C'est possible en créant un fournisseur qui lie les 2 fournisseurs ensemble :

Listing 13-25
```
1  # app/config/security.yml
2  security:
3      providers:
4          chain_provider:
5              chain :
6                  providers: [in_memory, user_db]
7          in_memory:
8              memory:
9                  users:
10                     foo: { password: test }
```

10. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle/Templating/GlobalVariables.html#getUser()

```
        user_db:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

Maintenant, tous les mécanismes d'authentification vont utiliser le `chain_provider`, car c'est le premier spécifié. Le `chain_provider` va essayer de charger les utilisateurs depuis les fournisseurs `in_memory` et `user_db`.

Si vous n'avez pas de raison de séparer vos utilisateurs `in_memory` des utilisateurs `user_db`, vous pouvez accomplir cela facilement en combinant les 2 sources dans un seul fournisseur :

Listing 13-26
```yaml
# app/config/security.yml
security:
    providers:
        main_provider:
            users:
                foo: { password: test }
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

Vous pouvez configurer le pare-feu ou des mécanismes individuels d'authentification afin qu'ils utilisent un fournisseur spécifique. Encore une fois, le premier fournisseur sera toujours utilisé, sauf si vous en spécifiez un explicitement :

Listing 13-27
```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            # ...
            provider: user_db
            http_basic:
                realm: "Secured Demo Area"
                provider: in_memory
            form_login: ~
```

Dans cet exemple, si un utilisateur essaie de se connecter via l'authentification HTTP, le système utilisera le fournisseur d'utilisateurs `in_memory`. Mais si l'utilisateur essaie de se connecter via le formulaire de connexion, le fournisseur `user_db` sera utilisé (car c'est celui par défaut du pare-feu).

Pour plus d'informations à propos des fournisseurs d'utilisateurs et de la configuration des pare-feu, veuillez vous reporter à *Configuration de référence de la Sécurité*.

Les rôles

La notion de « rôle » est au centre du processus d'autorisation. Chaque utilisateur se fait assigner un groupe de rôles et chaque ressource nécessite un ou plusieurs rôles. Si un utilisateur a les rôles requis, l'accès est accordé. Sinon, l'accès est refusé.

Les rôles sont assez simples, et sont en fait des chaînes de caractères que vous créez et utilisez au besoin (même si les rôles sont des objets en interne). Par exemple, si vous désirez limiter l'accès à la section d'administration du blog de votre site web, vous pouvez protéger cette section en utilisant un rôle `ROLE_BLOG_ADMIN`. Ce rôle n'a pas besoin d'être défini quelque part - vous n'avez qu'à commencer à l'utiliser.

Tous les rôles *doivent* commencer par le préfixe `ROLE_` afin d'être gérés par Symfony2. Si vous définissez vos propres rôles avec une classe `Role``dédiée (plus avancé), n'utilisez pas le préfixe ``ROLE_`.

Rôles hiérarchiques

Au lieu d'associer plusieurs rôles aux utilisateurs, vous pouvez définir des règles d'héritage de rôle en créant une hiérarchie de rôles :

Listing 13-28
```yaml
# app/config/security.yml
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

Dans la configuration ci-dessus, les utilisateurs avec le rôle `ROLE_ADMIN` vont aussi avoir le rôle `ROLE_USER`. Le rôle `ROLE_SUPER_ADMIN` a les rôles `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH` et `ROLE_USER` (hérité de `ROLE_ADMIN`).

Se déconnecter

Généralement, vous désirez aussi que vos utilisateurs puissent se déconnecter. Heureusement, le pare-feu peut prendre ça en charge automatiquement lorsque vous activez le paramètre de configuration `logout` :

Listing 13-29
```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            # ...
            logout:
                path:   /logout
                target: /
    # ...
```

Une fois que c'est configuré au niveau de votre pare-feu, un utilisateur qui accèdera à `/logout` (ou quelle que soit la configuration de `path` que vous avez) sera déconnecté. L'utilisateur sera redirigé à la page d'accueil (la valeur du paramètre `target`). Les 2 paramètres de configuration `path` et `target` ont comme valeur par défaut ce qui est défini ici. En d'autres termes, sauf si vous voulez les changer, vous pouvez les omettre complètement et ainsi réduire votre configuration :

Listing 13-30
```yaml
logout: ~
```

Veuillez noter que vous n'aurez *pas* à implémenter un contrôleur pour l'URL `/logout` car le pare-feu se charge de tout. Vous *devez* toutefois créer une route afin de l'utiliser pour générer l'URL :

Depuis Symfony 2.1, vous *devez* avoir une route qui correspond à votre chemin de déconnexion. Sans route, vous ne pourrez pas vous déconnecter.

Listing 13-31

```yaml
# app/config/routing.yml
logout:
    pattern:   /logout
```

Une fois qu'un utilisateur s'est déconnecté, il sera redirigé à l'URL définie par le paramètre `target` (par exemple `homepage`). Pour plus d'informations sur la configuration de la déconnexion, veuillez lire *Security Configuration Reference*.

Contrôle d'accès dans les templates

Si vous désirez vérifier dans un template si un utilisateur possède un rôle donné, utilisez la fonction helper intégrée :

Listing 13-32
```twig
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Supprimer</a>
{% endif %}
```

Si vous utilisez cette fonction et que vous ne vous trouvez pas à une URL pour laquelle un pare-feu est actif, une exception sera lancée. Encore une fois, c'est toujours une bonne idée d'avoir un pare-feu qui couvre toutes les URLs (comme c'est montré dans ce chapitre).

Contrôle d'accès dans les Contrôleurs

Si vous désirez vérifier dans un contrôleur si l'utilisateur courant possède un rôle, utilisez la méthode `isGranted` du contexte de sécurité:

Listing 13-33
```php
public function indexAction()
{
    // show different content to admin users
    if ($this->get('security.context')->isGranted('ROLE_ADMIN')) {
        // Load admin content here
    }
    // load other regular content here
}
```

Un pare-feu doit être actif, sinon une exception sera lancée lors de l'appel à la méthode `isGranted`. Référez-vous aux notes ci-dessus par rapport aux templates pour plus de détails.

« Usurper l'identité » d'un utilisateur

Parfois, il peut être utile de pouvoir passer d'un utilisateur à un autre sans avoir à se déconnecter et à se reconnecter (par exemple si vous êtes en train de débugguer ou de comprendre un bug qu'un utilisateur obtient, mais que vous ne pouvez pas reproduire). Cela peut être facilement réalisé en activant l'auditeur (listener) `switch_user` du pare-feu :

Listing 13-34

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            # ...
            switch_user: true
```

Pour changer d'utilisateur, il suffit d'ajouter à la chaîne de requête le paramètre `_switch_user` et le nom d'utilisateur comme valeur à l'URL en cours :

Listing 13-35
```
http://example.com/somewhere?_switch_user=thomas
```

Pour revenir à l'utilisateur initial, utilisez le nom d'utilisateur spécial `_exit`:

Listing 13-36
```
http://example.com/somewhere?_switch_user=_exit
```

Bien sûr, cette fonctionnalité ne doit être accessible qu'à un petit groupe d'utilisateurs. Par défaut, l'accès est limité aux utilisateurs ayant le rôle `ROLE_ALLOWED_TO_SWITCH`. Le nom de ce rôle peut être modifié grâce au paramètre `role`. Pour plus de sécurité, vous pouvez aussi changer le nom du paramètre de configuration grâce au paramètre``parameter``:

Listing 13-37
```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            // ...
            switch_user: { role: ROLE_ADMIN, parameter: _want_to_be_this_user }
```

Authentification sans état

Par défaut, Symfony2 s'appuie sur cookie (la Session) pour garder le contexte de sécurité d'un utilisateur. Mais si vous utilisez des certificats ou l'authentification HTTP par exemple, la persistance n'est pas nécessaire, car l'identité est disponible à chaque requête. Dans ce cas, et si vous n'avez pas besoin de sauvegarder quelque chose entre les requêtes, vous pouvez activer l'authentification sans état (stateless authentication), ce qui veut dire qu'aucun cookie ne sera jamais créé par Symfony2 :

Listing 13-38
```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            http_basic: ~
            stateless:  true
```

Si vous utilisez un formulaire de connexion, Symfony2 va créer un cookie même si vous avez configuré `stateless` à `true`.

Utilitaires

New in version 2.2: Les classes `StringUtils` et `SecureRandom` ont été ajoutées dans Symfony 2.2

Le composant de Sécurité Symfony est fourni avec un ensemble d'utilitaires pratiques liés à la sécurité. Ces utilitaires sont utilisés par Symfony, mais vous devriez aussi les utiliser pour résoudre les problèmes qu'ils traitent.

Comparer des chaines de caractères

Le temps pris pour comparer deux chaines de caractères dépend de leurs différences. Cela peut être utilisé par un attaquant lorsque les deux chaines représentent un mot de passe par exemple; cela s'appelle une *attaque temporelle*[11].

En interne, pour comparer deux mots de passe, Symfony utilise un algorithme en temps constant; vous pouvez utiliser la même stratégie dans votre propre code grâce à la classe *`StringUtils`*[12]:

Listing 13-39
```php
use Symfony\Component\Security\Core\Util\StringUtils;

// est-ce password1 est égal à password2 ?
$bool = StringUtils::equals($password1, $password2);
```

Générer un nombre aléatoire sécurisé

Chaque fois que vous avez besoin de générer un nombre aléatoire sécurisé, nous vous incitons fortement à utiliser la classe *`SecureRandom`*[13]:

Listing 13-40
```php
use Symfony\Component\Security\Core\Util\SecureRandom;

$generator = new SecureRandom();
$random = $generator->nextBytes(10);
```

La méthode *`nextBytes()`*[14] retourne une chaine de caractères numérique d'une longueur égale au nombre passé en argument (10 dans l'exemple ci-dessus).

La classe SecureRandom est plus efficace lorsque OpenSSL est installé, mais s'il n'est pas disponible, elle se rabat sur un algorithme interne qui a besoin d'un fichier pour l'alimenter. Contentez-vous de passer le nom du fichier en argument pour l'activer:

Listing 13-41
```php
$generator = new SecureRandom('/some/path/to/store/the/seed.txt');
$random = $generator->nextBytes(10);
```

Vous pouvez aussi accéder à une instance aléatoire sécurisée directement depuis le conteneur d'injection de dépendance de Symfony. Son nom est `security.secure_random`.

11. http://fr.wikipedia.org/wiki/Attaque_temporelle
12. http://api.symfony.com/master/Symfony/Component/Security/Core/Util/StringUtils.html
13. http://api.symfony.com/master/Symfony/Component/Security/Core/Util/SecureRandom.html
14. http://api.symfony.com/master/Symfony/Component/Security/Core/Util/SecureRandom.html#nextBytes()

Derniers mots

La sécurité peut être un problème complexe à résoudre correctement dans une application. Heureusement, le composant de sécurité de Symfony se base un modèle bien éprouvé basé sur l'*authentification* et l'*autorisation*. L'authentification, qui arrive toujours en premier, est prise en charge par le pare-feu dont la responsabilité est de déterminer l'identité des utilisateurs grâce à différentes méthodes (par exemple l'authentification HTTP, les formulaires de connexion, etc.). Dans le cookbook, vous trouverez des exemples d'autres méthodes pour prendre en charge l'authentification, incluant une manière d'implémenter la fonction de cookie « se souvenir de moi » (« remember me »),

Une fois l'utilisateur authentifié, la couche d'autorisation peut déterminer si l'utilisateur a accès ou non à des ressources spécifiques. Le plus souvent, des *rôles* sont appliqués aux URLs, classes ou méthodes et si l'utilisateur courant ne possède pas ce rôle, l'accès est refusé. La couche d'autorisation est toutefois beaucoup plus complexe, et suit un système de « vote » afin que plusieurs entités puissent déterminer si l'utilisateur courant devrait avoir accès à une ressource donnée.

Apprenez en plus sur la sécurité et sur d'autres sujets dans le cookbook.

Apprenez plus grâce au Cookbook

- *Forcer HTTP/HTTPS*
- *Blacklister des utilisateurs par adresse IP address grâce à un électeur personnalisé*
- *Liste d'accès de contrôle (ACLs)*
- *Comment ajouter la fonctionnalité de login « Se souvenir de moi »*

Chapter 14
Le Cache HTTP

L'essence d'une application web riche est d'être dynamique. Peu importe l'efficacité de votre application, le traitement d'une requête sera toujours plus important que l'envoi d'une page statique.

Pour la plupart des applications web, cela ne pose pas de problème. Symfony2 est d'une rapidité foudroyante, et à moins que vous ne fassiez de sérieux remaniements, chaque requête sera traitée rapidement sans trop « stresser » votre serveur.

Mais si la fréquentation de votre site augmente, ce traitement peut devenir un problème. Le processus qui s'effectue à chaque requête peut être exécuté une unique fois. C'est exactement l'objectif de la mise en cache.

La Mise en cache

Le moyen le plus efficace d'améliorer les performances d'une application est de mettre en cache l'intégralité d'une réponse pour ne plus avoir à rappeler l'application pour les requêtes suivantes. Bien sûr, ce n'est pas toujours possible pour les sites web fortement dynamiques, ou peut être que si. A travers ce chapitre, vous verrez comment fonctionne le système de cache de Symfony2 et en quoi c'est la meilleure approche possible.

Le système de cache de Symfony2 est différent, car il se base sur la simplicité et la puissance du cache HTTP tel qu'il est défini dans la *spécification HTTP*. Au lieu de réinventer un processus de mise en cache, Symfony2 adopte la norme qui définit la communication de base sur le Web. Une fois que vous avez compris les fondamentaux de la validation HTTP et de l'expiration de la mise en cache, vous serez prêt à maîtriser le système de cache de Symfony2.

Dans le but de comprendre comment mettre en cache avec Symfony, nous allons parcourir ce sujet en 4 étapes :

1. Une *passerelle de cache*, ou reverse proxy, est une couche indépendante qui se trouve devant votre application. La passerelle met en cache les réponses telles qu'elles sont retournées par l'application et répond aux requêtes avec les réponses qui sont en cache avant qu'elles n'atteignent l'application. Symfony2 possède sa propre passerelle par défaut, mais n'importe quelle autre peut être également utilisée.
2. Les entêtes du *cache HTTP* sont utilisés pour communiquer avec la passerelle de cache et tout autre cache entre votre application et le client. Symfony2 en propose par défaut et fournit une interface puissante pour interagir avec eux.

3. *L'expiration et la validation* HTTP sont les deux modèles utilisés pour déterminer si le contenu d'un cache est *valide* (peut être réutilisé à partir du cache) ou *périmé* (doit être régénéré par l'application).
4. Les *Edge Side Includes* (ESI) autorisent le cache HTTP à mettre en cache des fragments de pages (voir des fragments imbriqués) de façon indépendante. Avec les ESI, vous pouvez même mettre en cache une page entière pendant 60 minutes, mais un bloc imbriqué dans cette page uniquement 5 minutes.

Puisque la mise en cache via HTTP n'est pas spécifique à Symfony, de nombreux articles existent à ce sujet. Si vous n'êtes pas familier avec la mise cache HTTP, nous vous recommandons *fortement* de lire l'article de Ryan Tomayko *Things Caches Do*[1]. Le tutoriel de Mark Nottingham, *Cache Tutorial*[2], est également une ressource très complète sur ce sujet.

La mise en cache avec la Passerelle de Cache

Lors d'une mise en cache via HTTP, le *cache* est complètement séparé de votre application. Il est placé entre votre application et le client qui effectue les requêtes.

Le travail du cache est d'accepter les requêtes du client et de les transmettre à votre application. Le cache recevra aussi en retour des réponses de votre application et les enverra au client. Le cache est au milieu (« middle-man ») dans ce jeu de communication requête-réponse entre le client et votre application.

Lors d'une communication, le cache stockera toutes les réponses qu'ils estiment « stockables » (voir *Introduction à la mise en cache avec HTTP*). Si la même ressource est demandée, le cache renvoie le contenu mis en cache au client, en ignorant entièrement l'application.

Ce type de cache est connu sous le nom de passerelle de cache HTTP. Beaucoup d'autres solutions existent telles que *Varnish*[3], *Squid in reverse proxy mode*[4] et le reverse proxy de Symfony2.

Les types de caches

Mais une passerelle de cache ne possède pas qu'un seul type de cache. Les entêtes de cache HTTP envoyées par votre application sont interprétées par trois différents types de cache :

- *Le cache du navigateur* : tous les navigateurs ont leur propre cache qui est utile quand un utilisateur demande la page précédente ou des images et autres médias. Le cache du navigateur est un cache *privé*, car les ressources mises en cache ne sont partagées avec personne d'autre.
- *Le « cache proxy »* : un proxy est un cache *partagé* car plusieurs personnes peuvent être derrière un seul proxy. Il est habituellement installé par les entreprises et les FAIs pour diminuer le temps de réponse des sites et la consommation des ressources réseau.
- *Passerelle de cache* : comme un proxy, ce système de cache est également partagé, mais du côté serveur. Installé par des administrateurs réseau, il permet aux sites d'être plus extensibles, sûrs et performants.

Les passerelles de cache peuvent être désignées comme des « reverse proxy cache», « surrogate cache » ou même des accélérateurs HTTP.

1. http://tomayko.com/writings/things-caches-do
2. http://www.mnot.net/cache_docs/
3. https://www.varnish-cache.org/
4. http://wiki.squid-cache.org/SquidFaq/ReverseProxy

La notion de cache *privé* par rapport au cache *partagé* sera expliquée plus en détail lorsque la mise en cache des contenus liés à exactement un utilisateur (les informations sur un compte utilisateur par exemple) sera abordée.

Toutes les réponses de l'application iront communément dans un ou deux des deux premiers types de cache. Ces systèmes ne sont pas sous votre contrôle, mais suivent les directives du cache HTTP définies dans les réponses.

Symfony2 Reverse Proxy

Symfony2 contient un reverse proxy (aussi appelé passerelle de cache) écrit en PHP. Activez-le et les réponses de votre application qui peuvent être mise en cache seront immédiatement stockées. L'installer est aussi très simple. Chaque nouvelle application Symfony2 contient un noyau pré-configuré (AppCache) qui encapsule le noyau par défaut (AppKernel). Le cache kernel (cache du noyau) *est* le reverse proxy.

Pour activer le mécanisme de cache, il faut modifier le code du contrôleur principal pour qu'il utilise le cache kernel:

Listing 14-1
```php
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';
require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
// encapsule le AppKernel par défaut avec AppCache
$kernel = new AppCache($kernel);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

Le cache kernel se comportera immédiatement comme un « reverse proxy » en mettant en cache les réponses de l'application et en les renvoyant au client.

Le cache kernel a une méthode spéciale `getLog()` qui retourne une chaîne de caractères décrivant ce qui se passe dans la couche du cache. Dans l'environnement de développement, il est possible de l'utiliser pour du débogage ou afin de valider votre stratégie de mise en cache:

Listing 14-2
```php
error_log($kernel->getLog());
```

L'objet `AppCache` a une configuration par défaut, mais peut être reconfiguré finement grâce à une série d'options que vous pouvez paramétrer en surchargeant la méthode *`getOptions()`*[5]:

Listing 14-3
```php
// app/AppCache.php
use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;

class AppCache extends HttpCache
```

5. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle/HttpCache/HttpCache.html#getOptions()

```
5  {
6      protected function getOptions()
7      {
8          return array(
9              'debug'                 => false,
10             'default_ttl'           => 0,
11             'private_headers'       => array('Authorization', 'Cookie'),
12             'allow_reload'          => false,
13             'allow_revalidate'      => false,
14             'stale_while_revalidate' => 2,
15             'stale_if_error'        => 60,
16         );
17     }
18 }
```

A moins d'être surchargée dans `getOptions()`, l'option `debug` est par défaut égale à celle de l'objet `AppKernel` encapsulé.

Voici une liste des principales options :

- `default_ttl` : Le nombre de secondes pendant lesquelles une entrée du cache doit être considérée comme « valide » quand il n'y a pas d'information explicite fournie dans une réponse. Une valeur explicite pour les entêtes `Cache-Control` ou `Expires` surcharge cette valeur (par défaut : `0`);
- `private_headers` : Ensemble d'entêtes de requête qui déclenche le comportement « privé » du `Cache-Control` pour les réponses qui ne précisent pas explicitement si elle sont `publiques` ou `privées` via une directive du `Cache-Control`. (par défaut : `Authorization` et `Cookie`);
- `allow_reload` : Définit si le client peut forcer ou non un rechargement du cache en incluant une directive du `Cache-Control` « no-cache » dans la requête. Définissez-la à `true` pour la conformité avec la RFC 2616 (par défaut : `false`);
- `allow_revalidate` : Définit si le client peut forcer une revalidation du cache en incluant une directive de `Cache-Control` « max-age=0 » dans la requête. Définissez-la à `true` pour la conformité avec la RFC 2616 (par défaut : `false`);
- `stale_while_revalidate` : Spécifie le nombre de secondes par défaut (la granularité est la seconde parce que le TTL de la réponse est en seconde) pendant lesquelles le cache peut renvoyer une réponse « périmée » alors que la nouvelle réponse est calculée en arrière-plan (par défaut : `2`). Ce paramètre est surchargé par l'extension HTTP `stale-while-revalidate` du `Cache-Control` (cf. RFC 5861);
- `stale_if_error` : Spécifie le nombre de secondes par défaut (la granularité est la seconde) pendant lesquelles le cache peut renvoyer une réponse « périmée » quand une erreur est rencontrée (par défaut : `60`). Ce paramètre est surchargé par l'extension HTTP `stale-if-error` du `Cache-Control` (cf. RFC 5961).

Si le paramètre `debug` est à `true`, Symfony2 ajoute automatiquement l'entête `X-Symfony-Cache` à la réponse contenant des informations utiles à propos des caches « hits » (utilisation du cache) et « misses » (page ou réponse non présente en cache).

Passer d'un Reverse Proxy à un autre

Le reverse proxy de Symfony2 est un formidable outil lors de la phase de développement de votre site web ou lors d'un déploiement sur des serveurs mutualisés sur lesquels il n'est pas possible d'installer d'autres outils que ceux proposés par PHP. Mais il n'est pas aussi performant que des proxy écrits en C. C'est pourquoi il est fortement recommandé d'utiliser Varnish ou Squid sur les serveurs de production si possible. La bonne nouvelle est qu'il est très simple de passer d'un proxy à un autre sans qu'aucune modification ne soit nécessaire dans le code. Vous pouvez commencez avec le reverse proxy de Symfony2 puis le mettre à jour plus tard vers Varnish quand votre trafic augmentera.

Pour plus d'informations concernant Varnish avec Symfony2, veuillez- vous reportez au chapitre du cookbook *Comment utiliser Varnish*.

Les performances du reverse proxy de Symfony2 ne sont pas liées à la complexité de votre application. C'est parce que le noyau de l'application n'est démarré que quand la requête lui est transmise.

Introduction à la mise en cache avec HTTP

Pour tirer parti des couches de gestion du cache, l'application doit être capable de communiquer quelles réponses peuvent être mises en cache et les règles, qui décident quand et comment le cache devient obsolète. Cela se fait en définissant des entêtes de cache HTTP dans la réponse.

Il faut garder à l'esprit que « HTTP » n'est rien d'autre que le langage (un simple langage texte) que les clients web (les navigateurs par exemple) et les serveurs utilisent pour communiquer entre eux. Parler de mise en cache HTTP revient à parler de la partie du langage qui permet aux clients et aux serveurs d'échanger les informations relatives à la gestion du cache.

HTTP définit quatre entêtes de cache que nous détaillons ici :

- `Cache-Control`
- `Expires`
- `ETag`
- `Last-Modified`

L'entête le plus important et le plus versatile est l'entête `Cache-Control` qui est en réalité une collection d'informations diverses sur le cache.

Tous ces entêtes seront complètement détaillés dans la section *HTTP Expiration et Validation*.

L'entête Cache-Control

Cet entête est unique du fait qu'il contient non pas une, mais un ensemble varié d'informations sur la possibilité de mise en cache d'une réponse. Chaque information est séparée par une virgule :

```
Cache-Control: private, max-age=0, must-revalidate
```

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

Symfony fournit une abstraction du `Cache-Control` pour faciliter sa gestion:

Listing 14-4
```php
// ...

use Symfony\Component\HttpFoundation\Response;

$response = new Response();

// marque la réponse comme publique ou privée
$response->setPublic();
$response->setPrivate();

// définit l'âge max des caches privés ou des caches partagés
$response->setMaxAge(600);
$response->setSharedMaxAge(600);

 // définit une directive personnalisée du Cache-Control
$response->headers->addCacheControlDirective('must-revalidate', true);
```

Réponse publique et réponse privée

Les passerelles de cache et les caches « proxy » sont considérés comme étant « partagés » car leur contenu est partagé par plusieurs utilisateurs. Si une réponse spécifique à un utilisateur est stockée par erreur dans ce type de cache, elle pourrait être renvoyée à plusieurs autres utilisateurs. Imaginez si les informations concernant votre compte sont mises en cache et ensuite envoyées à tous les utilisateurs suivants qui souhaitent accéder à leur page de compte !

Pour gérer cette situation, chaque réponse doit être définie comme étant publique ou privée :

- ***public*: Indique que la réponse peut être mise en cache, à la fois,**

  par les caches privés et les caches publics;

- ***private*: Indique que toute la réponse concerne un unique**

  utilisateur et qu'elle ne doit pas être stockée dans les caches publics.

Symfony considère par défaut chaque réponse comme étant privée. Pour tirer parti des caches partagés (comme le reverse proxy de Symfony2), la réponse devra explicitement être définie comme publique.

Méthodes sures

La mise en cache HTTP ne fonctionne qu'avec les méthodes « sures » (telles que GET et HEAD). « Être sûr » signifie que l'état de l'application n'est jamais modifié par le serveur au moment de servir la requête (il est bien sûr possible de loguer des informations, mettre en cache des données, etc.). Cela a deux conséquences :

- L'état de l'application ne devrait *jamais* être modifié en répondant à une requête GET ou HEAD. Même s'il n'y a pas de passerelle de cache, la présence d'un cache « proxy » signifie qu'aucune requête GET ou HEAD ne pourrait pas atteindre le serveur.
- Ne pas mettre en cache les méthodes PUT, POST ou DELETE. Ces méthodes sont normalement utilisées pour changer l'état de l'application (supprimer un billet de blog par exemple). La mise en cache de ces méthodes empêcherait certaines requêtes d'atteindre et de modifier l'application.

Règles de mise en cache et configuration par défaut

HTTP 1.1 permet de tout mettre en cache par défaut à moins qu'il n'y ait un entête `Cache-Control` explicite. En pratique, la plupart des systèmes de cache ne font rien quand les requêtes contiennent un cookie, ont un entête d'autorisation, utilisent une méthode non sure (i.e. PUT, POST, DELETE), ou quand les réponses ont un code de redirection.

Symfony2 définit automatiquement une configuration de l'entête Cache-Control, quand aucun n'est défini par le développeur, en suivant ces règles :

- Si aucun entête de cache n'est défini (`Cache-Control`, `Expires`, `ETag` ou `Last-Modified`), `Cache-Control` est défini à `no-cache`, ce qui veut dire que la réponse ne sera pas mise en cache;
- Si `Cache-Control` est vide (mais que l'un des autres entêtes de cache est présent) sa valeur est définie à `private, must-revalidate`;
- Mais si au moins une directive `Cache-Control` est définie, et qu'aucune directive `publique` ou `private` n'a été ajoutée explicitement, Symfony2 ajoute la directive `private` automatiquement (sauf quand `s-maxage` est défini).

HTTP Expiration et Validation

La spécification HTTP définit deux modèles de mise en cache :

- Avec le *modèle d'expiration*[6], on spécifie simplement combien de temps une réponse doit être considérée comme « valide » en incluant un entête `Cache-Control` et/ou `Expires`. Les systèmes de cache qui supportent l'expiration enverront la même réponse jusqu'à ce que la version en cache soit expirée et devienne « invalide ».
- Quand une page est dynamique (c-a-d quand son contenu change souvent), le *modèle de validation*[7] est souvent nécessaire. Avec ce modèle, le système de cache stocke la réponse, mais demande au serveur à chaque requête si la réponse est encore valide. L'application utilise un identifiant unique (l'entête `Etag`) et/ou un timestamp (l'entête `Last-Modified`) pour vérifier si la page a changé depuis sa mise en cache.

Le but de ces deux modèles est de ne jamais générer deux fois la même réponse en s'appuyant sur le système de cache pour stoker et renvoyer la réponse valide.

En lisant la spécification HTTP

La spécification HTTP définit un langage simple, mais puissant dans lequel les clients et les serveurs peuvent communiquer. En tant que développeur web, le modèle requête-réponse est le plus populaire. Malheureusement, le document de spécification - *RFC 2616*[8] - peut être difficile à lire.

Il existe actuellement une tentative (*HTTP Bis*[9]) de réécriture du RFC 2616. Elle ne décrit pas une nouvelle version du HTTP mais clarifie plutôt la spécification originale du HTTP. Elle est découpée en sept parties ; tout ce qui concerne la gestion du cache se retrouve dans deux chapitres dédiés (P4 - Conditional Requests et *P6 - Caching: Browser and intermediary caches*[10]).

En tant que développeur web, il est fortement recommandé de lire la spécification. Sa clarté et sa puissance - même plus dix ans après sa création - est inestimable. Ne soyez pas rebuté par l'apparence du document - son contenu est beaucoup plus intéressant que son aspect.

6. http://tools.ietf.org/html/rfc2616#section-13.2
7. http://tools.ietf.org/html/rfc2616#section-13.3
8. http://tools.ietf.org/html/rfc2616
9. http://tools.ietf.org/wg/httpbis/
10. http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-12

Expiration

Le modèle d'expiration du cache est le plus efficace et le plus simple à mettre en place et devrait être utilisé dès que possible. Quand une réponse est mise en cache avec une directive d'expiration, le cache stockera la réponse et la renverra directement sans solliciter l'application avant son expiration.

Ce modèle est mis en oeuvre avec deux entêtes HTTP presque identiques : `Expires` ou `Cache-Control`.

Expiration avec l'entête `Expires`

D'après la spécification HTTP, « le champ de l'entête `Expires` donne la date ou le temps après laquelle la réponse est considérée comme invalide ». L'entête `Expires` peut être défini avec la méthode `setExpires()` de l'objet `Response`. Elle prend un objet `DateTime` en argument:

Listing 14-5
```php
$date = new DateTime();
$date->modify('+600 seconds');

$response->setExpires($date);
```

L'entête HTTP résultante sera :

Listing 14-6
```
Expires: Thu, 01 Mar 2011 16:00:00 GMT
```

La méthode `setExpires()` convertit automatiquement la date au format GMT comme demandé par la spécification.

Notez que dans toutes les versions HTTP précédant la 1.1, le serveur d'origine n'était pas obligé d'envoyer l'entête `Date`. En conséquence, le cache (par exemple le navigateur) pourrait être obligé de consulter l'horloge locale afin d'évaluer l'entête `Expires` rendant ainsi le calcul de la durée de vie sensible aux décalages horaires. Une autre limitation de l'entête `Expires` est que la spécification déclare que « les serveurs HTTP/1.1 ne devraient pas envoyer des dates `Expires` de plus d'un an dans le futur ».

Expiration avec l'entête `Cache-Control`

À cause des limitations de l'entête `Expires`, bien souvent, il faut utiliser l'entête `Cache-Control`. Rappelez-vous que l'entête `Cache-Control` est utilisé pour spécifier une grande partie des directives de cache. Pour le modèle d'expiration, il y a deux directives, `max-age` et `s-maxage`. La première est utilisée par tous les systèmes de cache alors que la seconde n'est utilisée que par les systèmes de cache partagés:

Listing 14-7
```php
// Définir le nombre de secondes après lesquelles la réponse
// ne devrait plus être considérée comme valide
$response->setMaxAge(600);

// Idem mais uniquement pour les caches partagés
$response->setSharedMaxAge(600);
```

L'entête `Cache-Control` devrait être (il peut y avoir d'autres directives):

Listing 14-8
```
Cache-Control: max-age=600, s-maxage=600
```

Validation

S'il faut mettre à jour une ressource dès qu'il y a un changement de données, le modèle d'expiration ne convient pas. Avec le modèle d'expiration, l'application ne sera pas appelée jusqu'au moment où le cache devient invalide.

Le modèle de validation du cache corrige ce problème. Dans ce modèle, le cache continue de stocker les réponses. La différence est que pour chaque requête, le cache demande à l'application si la réponse en cache est encore valide. Si la réponse en cache *est* toujours valide, l'application renvoie un code statut 304 et aucun contenu. Cela indique au cache qu'il est autorisé à renvoyer la réponse mis en cache.

Ce modèle permet d'économiser beaucoup de bande passante, car la même réponse n'est pas envoyée deux fois au même client (un code 304 est envoyé à la place). Si l'application est bien construite, il est possible de déterminer le minimum de données nécessitant l'envoi de réponse 304 et aussi d'économiser des ressources CPU (voir ci-dessous pour un exemple d'implémentation).

Le code 304 signifie « Not Modified » (non modifié). C'est important, car avec ce code statut, la réponse ne contient *pas* le contenu demandé. Au lieu de cela, la réponse est simplement un ensemble léger de directives qui informe le cache qu'il devrait utiliser la réponse stockée.

Comme avec le modèle d'expiration, il y a deux différents types d'entêtes HTTP qui peuvent être utilisés pour implémenter ce modèle : `ETag` et `Last-Modified`.

Validation avec l'entête `ETag`

L'entête `ETag` est une chaîne de caractères (appelée « entity-tag ») qui identifie de façon unique une représentation de la ressource appelée. Il est entièrement généré et défini par votre application de telle sorte que vous pouvez spécifier, par exemple, si la ressource `/about`, stockée en cache, est à jour avec ce que votre application retourne. Un `ETag` est similaire à une empreinte et est utilisé pour comparer rapidement si deux versions différentes d'une ressource sont équivalentes. Comme une empreinte, chaque `ETag` doit être unique pour toutes les représentations de la même ressource.

Voici une implémentation simple qui génère l'entête ETag depuis un md5 du contenu:

Listing 14-9
```php
public function indexAction()
{
    $response = $this->render('MyBundle:Main:index.html.twig');
    $response->setETag(md5($response->getContent()));
    $response->setPublic(); // permet de s'assurer que la réponse est publique, et qu'elle
peut donc être cachée
    $response->isNotModified($this->getRequest());

    return $response;
}
```

La méthode *`isNotModified()`*[11] method:*Symfony\Component\HttpFoundation\Response::isNotModified* compare le `ETag` envoyé avec la requête avec celui défini dans l'objet `Response`. S'ils sont identiques, la méthode définit automatiquement le code de l'objet `Response` comme 304.

Cet algorithme est assez simple et très générique, mais il implique de créer entièrement l'objet `Response` avant de pouvoir calculer l'entête ETag, ce qui n'est pas optimal. En d'autres termes, cette approche économise la bande passante, mais pas l'utilisation du CPU.

Dans la section *Optimiser son code avec le modèle de validation du cache*, vous verrez comment le modèle de validation peut être utilisé plus intelligemment pour déterminer la validité d'un cache sans faire autant de travail.

11. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html#isNotModified()

Symfony2 supporte aussi les ETags moins robustes en définissant le second argument à `true` pour la méthode *`setETag()`*[12].

Validation avec l'entête `Last-Modified`

L'entête `Last-Modified` est la seconde forme de validation. D'après la spécification HTTP, « le champ de l'entête `Last-Modified` indique la date et l'heure à laquelle le serveur d'origine croit que la représentation a été modifiée pour la dernière fois ». En d'autres termes, l'application décide si oui ou non le contenu du cache a été mis à jour, en se basant sur le fait que, si oui ou non le cache a été mis à jour depuis que la réponse a été mise en cache.

Par exemple, vous pouvez utiliser la date de dernière mise à jour de tous les objets nécessitant de calculer le rendu de la ressource comme valeur de l'entête `Last-Modified`:

Listing 14-10

```php
public function showAction($articleSlug)
{
    // ...

    $articleDate = new \DateTime($article->getUpdatedAt());
    $authorDate = new \DateTime($author->getUpdatedAt());

    $date = $authorDate > $articleDate ? $authorDate : $articleDate;

    $response->setLastModified($date);
     // Définit la réponse comme publique. Sinon elle sera privée par défaut.
    $response->setPublic();

    if ($response->isNotModified($this->getRequest())) {
        return $response;
    }

    // ... ajoutez du code ici pour remplir la réponse avec le contenu complet

    return $response;
}
```

La méthode *`isNotModified()`*[13] compare l'entête `If-Modified-Since` envoyé par la requête avec l'entête `Last-Modified` défini dans la réponse. S'ils sont équivalents, l'objet `Response` aura un code status 304.

L'entête de la requête `If-Modified-Since` est égal à l'entête de la dernière réponse `Last-Modified` du client pour une ressource donnée. C'est grâce à cela que le client et le serveur communiquent et constatent ou non si la ressource a été mise à jour depuis qu'elle est en cache.

Optimiser son code avec le modèle de validation du cache

Le but principal de toutes les stratégies de mise en cache est de diminuer la charge de l'application. Autrement dit, moins l'application aura à « travailler » pour renvoyer un status 304, mieux ce sera. La méthode `Response::isNotModified()` fait exactement ça en exposant un modèle simple et efficace:

Listing 14-11

12. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html#setETag()
13. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html#isNotModified()

```php
use Symfony\Component\HttpFoundation\Response;

public function showAction($articleSlug)
{
    // Récupère le minimum d'informations pour calculer
    // l'ETag ou la dernière valeur modifiée (Last-Modified value)
    // (basé sur l'objet Request, les données sont recueillies
    // d'une base de données ou d'un couple clé-valeur
    // par exemple)
    $article = ...;

    // Crée un objet Response avec un entête ETag
    // et/ou un entête Last-Modified
    $response = new Response();
    $response->setETag($article->computeETag());
    $response->setLastModified($article->getPublishedAt());

    // Définit la réponse comme publique. Sinon elle sera privée par défaut.
    $response->setPublic();

    // Vérifie que l'objet Response n'est pas modifié
    // pour un objet Request donné
    if ($response->isNotModified($this->getRequest())) {
        // Retourne immédiatement un objet 304 Response
        return $response;
    } else {
        // faire plus de travail ici - comme récupérer plus de données
        $comments = ...;

        // ou formatter un template avec la $response déjà existante
        return $this->render(
            'MyBundle:MyController:article.html.twig',
            array('article' => $article, 'comments' => $comments),
            $response
        );
    }
}
```

Quand l'objet `Response` n'est pas modifié, la méthode `isNotModified()` définit automatiquement le code 304, enlève le contenu et les entêtes qui ne doivent pas être présents pour un status `304` (voir la *`setNotModified()`*[14]).

Faire varier la Response

Jusqu'ici, nous avons supposé que chaque URI est une représentation unique de la ressource cible. Par défaut, la mise en cache HTTP est faite en donnant l'URI de la ressource comme clé de cache. Si deux personnes demandent la même URI d'une ressource qui peut être mise en cache, la deuxième personne recevra la version qui est dans le cache.

Dans certains cas, ce n'est pas suffisant et des versions différentes de la même URI ont besoin d'être mises en cache en fonction des valeurs d'un ou plusieurs entêtes. Par exemple, si les pages sont compressées parce que le client le supporte, n'importe quelle URI a deux représentations : une quand le client accepte la compression, l'autre quand le client ne l'accepte pas. Cette détermination est faite grâce à la valeur de l'entête `Accept-Encoding`.

Dans ce cas, le cache doit contenir une version compressée et une version non compressée de la réponse pour une URI particulière et les envoyer en fonction de la valeur `Accept-Encoding` de la requête. Cela

14. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html#setNotModified()

est possible en utilisant l'entête `Vary` de la réponse, qui est une liste des différents entêtes séparés par des virgules dont les valeurs définissent une représentation différente de la même ressource.

Listing 14-12
```
Vary: Accept-Encoding, User-Agent
```

Cet entête `Vary` particulier permettra la mise en cache de versions différentes de la même ressource en se basant sur l'URI et la valeur des entêtes `Accept-Encoding` et `User-Agent`.

L'objet `Response` propose une interface pour gérer l'entête `Vary`:

Listing 14-13
```php
// définit un entête "vary"
$response->setVary('Accept-Encoding');

// définit plusieurs entêtes "vary"
$response->setVary(array('Accept-Encoding', 'User-Agent'));
```

La méthode `setVary()` prend un nom d'entête ou un tableau de noms d'entête pour lesquels la réponse varie.

Expiration et Validation

Il est, bien entendu, possible d'utiliser à la fois le modèle de validation et d'expiration pour un même objet `Response`. Mais comme le modèle d'expiration l'emporte sur le modèle de validation, il est facile de bénéficier du meilleur des deux modèles. En d'autres termes en utilisant à la fois l'expiration et la validation, vous pouvez programmer le cache pour qu'il fournisse son contenu pendant qu'il vérifie à intervalle régulier (l'expiration) que ce contenu est toujours valide.

Les autres méthodes de l'objet Response

La classe Response fournit beaucoup d'autres méthodes en relation avec la gestion du cache. Voici les plus utiles:

Listing 14-14
```php
// Marque l'objet Response comme obsolète
$response->expire();

// Forcer le retour d'une réponse 304 sans aucun contenu
$response->setNotModified();
```

La plupart des entêtes en relation avec la gestion du cache peuvent être définis avec la seule méthode *`setCache()`*[15]:

Listing 14-15
```php
// Définit la configuration du cache en un seul appel
$response->setCache(array(
    'etag'          => $etag,
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
    'public'        => true,
    // 'private'    => true,
));
```

15. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html#setCache()

Utilisation de la technologie « Edge Side Includes »

Les passerelles de caches sont une bonne solution pour améliorer les performances d'un site. Mais elles ont une limitation : elles peuvent uniquement mettre en cache une page dans son intégralité. S'il n'est pas possible de mettre une page entière en cache ou si des parties de cette page sont plus dynamiques que d'autres, cela pose problème. Heureusement, Symfony2 fournit une solution pour ces situations, basée sur la technologie « Edge Side Includes », aussi appelée *ESI*[16]. Akamaï a écrit cette spécification il y a 10 ans ; elle permet de mettre en cache une partie de page avec une stratégie différente de l'ensemble de la page.

La spécification « ESI » décrit des marqueurs (« tags ») qui peuvent être embarqués dans la page pour communiquer avec la passerelle de cache. Un seul marqueur est implémenté dans Symfony2, `include` car c'est le seul qui est utile en dehors du contexte Akamaï :

Listing 14-16
```
<!DOCTYPE html>
<html>
    <body>
        <!-- ... du contenu -->

        <!-- inclut le contenu d'une autre page ici -->
        <esi:include src="http://..." />

        <!-- ... du contenu -->
    </body>
</html>
```

L'exemple montre que chaque marqueur ESI a une URL complète (fully-qualified). Un marqueur ESI représente un morceau de page qui peut être appelé via une URL donnée.

Quand une requête est envoyée, la passerelle de cache appelle la page entière depuis son espace de stockage ou depuis le « backend » de l'application. Si la réponse contient un ou plusieurs marqueurs ESI, ils sont gérés de la même manière. En d'autres termes, la passerelle de cache récupère les fragments de page de son cache, ou demande à l'application de les recalculer. Quand tous les marqueurs ont été calculés, la passerelle les « fusionne » avec la page principale et envoie le contenu final vers le client.

Le processus est géré de manière transparente au niveau de la passerelle de cache (c-a-d à l'extérieur de l'application). Comme vous pouvez le voir, si vous décidez de prendre l'avantage des marqueurs ESI, Symfony2 réalise le procédé pour les inclure presque sans effort.

Utiliser ESI avec Symfony2

Premièrement, pour utiliser ESI, il faut l'activer dans la configuration de l'application :

Listing 14-17
```
# app/config/config.yml
framework:
    # ...
    esi: { enabled: true }
```

Maintenant, prenons l'exemple d'une page statique, excepté pour l'espace « Actualités » qui se trouve en bas de page. Avec ESI, il est possible de mettre en cache la partie qui gère les actualités indépendamment du reste de la page.

Listing 14-18

16. http://www.w3.org/TR/esi-lang

```php
public function indexAction()
{
    $response = $this->render('MyBundle:MyController:index.html.twig');
    // définit l'âge maximal partagé - cela marque aussi la réponse comme étant publique
    $response->setSharedMaxAge(600);

    return $response;
}
```

Dans cet exemple, la page a une espérance de vie de 10 minutes en cache. Dans un deuxième temps, incluons l'élément relatif à l'actualité dans un template via une action embarquée. Ceci sera réalisé grâce au « helper » `render` (voir la documentation sur *Contrôleurs imbriqués* pour plus de détails).

Comme le contenu embarqué provient d'une autre page (ou d'un autre contrôleur), Symfony2 utilise le « helper » standard `render` pour configurer le marqueur ESI :

Listing 14-19
```twig
{# vous pouvez utiliser une référence de contrôleur #}
{{ render_esi(controller('...:news', { 'max': 5 })) }}

{# ... ou une URL #}
{{ render_esi(url('latest_news', { 'max': 5 })) }}
```

En utilisant le rendu `esi` (via la fonction Twig `render_esi`), vous spécifiez à Symfony que l'action doit être retournée dans une balise ESI. Vous vous demandez peut être pourquoi il est préférable d'utiliser un helper plutôt que d'afficher la balise ESI vous même. C'est parce qu'en utilisant un helper, vous êtes sur que votre application fonctionnera même si aucune passerelle de cache n'est installée.

Lorsque vous utilisez la fonction `render` (ou en définissant le rendu à `inline`), Symfony2 merge le contenu de la page incluse dans la page principale avant d'envoyer la réponse au client. Mais si vous utilisez le rendu `esi` (c-a-d en appelant `render_esi`), *et* si Symfony2 détecte une passerelle de cache qui supporte ESI, alors une balise include ESI est générée. Mais s'il y a aucune passerelle de cache, ou si elle ne supporte pas ESI, Symfony2 mergera le contenu de la page incluse dans la page principale, comme si vous aviez appelé `render`.

Symfony2 détecte si la passerelle gère les marqueurs ESI grâce à une autre spécification de Akamaï qui est d'ores et déjà supporté par le reverse proxy de Symfony2.

L'action incluse peut maintenant spécifier ces propres règles de gestion du cache, entièrement indépendamment du reste de la page.

Listing 14-20
```php
public function newsAction($max)
{
    // ...

    $response->setSharedMaxAge(60);
}
```

Avec ESI, la page complète sera valide pendant 600 secondes, mais le composant de gestion des actualités ne le sera que pendant 60 secondes.

Lorsque vous utilisez une référence de contrôleur, le tag ESI doit pouvoir appeler l'action incluse via une URL accessible pour que la passerelle de cache puisse la recharger indépendamment du reste de la page. Symfony2 se charge de générer une URL unique pour chaque référence de contrôleur et est capable de les router correctement grâce à un écouteur qui doit être activé dans votre configuration:

Listing 14-21
```yaml
# app/config/config.yml
framework:
    # ...
    fragments: { path: /_fragment }
```

Un des grands avantages de cette stratégie de cache est qu'il est possible d'avoir une application aussi dynamique que souhaité tout en faisant appel à cette application le moins possible.

L'écouteur ne répond qu'à des adresses IP locale ou a des proxys de confiance.

Une fois que ESI est utilisée, il ne faut pas oublier de toujours utiliser la directive `s-maxage` à la place de `max-age`. Comme le navigateur ne reçoit que la réponse « agrégée » de la ressource, il n'est pas conscient de son « sous-contenu », il suit la directive `max-age` et met toute la page en cache. Et ce n'est pas ce que vous voulez.

Le helper `render_esi` supporte deux autres options utiles :

- `alt`: utilisée comme l'attribut `alt` du marqueur ESI, il permet de spécifier une URL alternative si la ressource `src` ne peut pas être trouvée ;
- `ignore_errors`: s'il est défini à `true`, un attribut `onerror` sera ajouté à l'ESI avec une valeur `continue` indiquant que, en cas d'échec, la passerelle de cache enlèvera le marqueur ESI sans erreur ou warning.

Invalidation du cache

> « There are only two hard things in Computer Science: cache invalidation and naming things. » --Phil Karlton
>
> Ceci peut être traduit comme : « Il existe uniquement deux opérations délicates en Informatique : l'invalidation de cache et nommer les choses. »

Vous ne devriez jamais avoir besoin d'invalider des données du cache parce que l'invalidation est déjà prise en compte nativement par le modèle de gestion du cache HTTP. Si la validation est utilisée, vous ne devriez pas avoir besoin d'utiliser l'invalidation par définition ; si l'expiration est utilisée et que vous avez besoin d'invalider une ressource, c'est que date d'expiration a été définie trop loin dans le futur.

Puisque l'invalidation est un sujet spécifique à chaque type de reverse proxy, si vous ne vous occupez pas de l'invalidation, vous pouvez passer d'un reverse proxy à l'autre sans changer quoi que ce soit au code de votre application.

En fait, tous les « reverse proxies » fournissent un moyen de purger les données du cache mais il faut l'éviter autant que possible. Le moyen le plus standard est de purger le cache pour une URL donnée en l'appelant avec la méthode HTTP spéciale `PURGE`.

Voici comment configurer le reverse proxy de Symfony2 pour supporter méthode HTTP ``PURGE`:

Listing 14-22
```php
// app/AppCache.php

// ...
```

```php
use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

class AppCache extends HttpCache
{
    protected function invalidate(Request $request, $catch = false)
    {
        if ('PURGE' !== $request->getMethod()) {
            return parent::invalidate($request, $catch);
        }

        $response = new Response();
        if (!$this->getStore()->purge($request->getUri())) {
            $response->setStatusCode(404, 'Not purged');
        } else {
            $response->setStatusCode(200, 'Purged');
        }

        return $response;
    }
}
```

Il faut protéger cette méthode HTTP `PURGE` d'une manière ou d'une autre pour éviter que n'importe qui ne puisse purger le cache.

Résumé

Symfony2 a été conçu pour suivre les règles éprouvées du protocole HTTP. La mise en cache n'y fait pas exception. Comprendre le système de cache de Symfony2 revient à bien comprendre les modèles de gestion du cache HTTP et à les utiliser efficacement. Ceci veut dire qu'au lieu de vous appuyer uniquement sur la documentation et les exemples de code de Symfony2, vous pouvez vous ouvrir à un monde plein de connaissances relatives au cache et passerelles de cache HTTP telles que Varnish.

En savoir plus grâce au Cookbook

- *Comment utiliser Varnish pour accélérer mon site Web*

Chapter 15
Traductions

Le terme « internationalisation » (souvent abrégé *i18n*[1]) désigne le processus d'abstraction des textes et autres spécificités locales en dehors de votre application qui sont ensuite placés dans un fichier où ils peuvent être traduits et convertis en se basant sur la locale de l'utilisateur (i.e. la langue et le pays). Pour du texte, cela signifie l'encadrer avec une fonction capable de traduire le texte (ou « message ») dans la langue de l'utilisateur :

Listing 15-1
```php
// le texte va *toujours* s'afficher en anglais
echo 'Hello World';

// le texte peut être traduit dans la langue de l'utilisateur final ou par défaut en anglais
echo $translator->trans('Hello World');
```

Le terme *locale* désigne en gros la langue et le pays de l'utilisateur. Cela peut être n'importe quelle chaîne de caractères que votre application va utiliser pour gérer les traductions et autres différences de format (par ex. format de monnaie). Le code *langue ISO639-1*[2], un « underscore » (_), et ensuite le code *pays ISO3166 Alpha-2*[3] (par ex. `fr_FR` pour Français/France) est recommandé.

Dans ce chapitre, vous apprendrez comment préparer une application à gérer de multiples locales et ensuite comment créer des traductions pour plusieurs locales. Dans l'ensemble, le processus a plusieurs étapes communes :

1. Activer et configurer le composant `Translation` de Symfony ;
2. Faire abstraction des chaînes de caractères (i.e. « messages ») en les encadrant avec des appels au `Translator` ;
3. Créer des ressources de traduction pour chaque locale supportée qui traduit chaque message dans l'application ;
4. Déterminer, définir et gérer la locale de l'utilisateur dans la requête, et optionnellement dans la session.

1. http://fr.wikipedia.org/wiki/Internationalisation_et_localisation
2. http://fr.wikipedia.org/wiki/Liste_des_codes_ISO_639-1
3. http://fr.wikipedia.org/wiki/ISO_3166-1#Table_de_codage

Configuration

Les traductions sont traitées par le *service* `Translator` qui utilise la locale de l'utilisateur pour chercher et retourner les messages traduits. Avant de l'utiliser, activez le `Translator` dans votre configuration :

Listing 15-2
```yaml
# app/config/config.yml
framework:
    translator: { fallback: en }
```

L'option `fallback` définit la locale de secours à utiliser quand une traduction n'existe pas dans la locale de l'utilisateur.

Quand une traduction n'existe pas pour une locale donnée, le traducteur (« Translator ») essaye tout d'abord de trouver une traduction pour cette langue (`fr` si la locale est `fr_FR` par exemple). Si cela échoue également, il regarde alors pour une traduction utilisant la locale de secours.

La locale utilisée pour les traductions est celle qui est stockée dans la requête. Vous pouvez la définir grâce à l'attribut `_locale` de vos routes (*La locale et l'URL*).

Traduction basique

La traduction du texte est faite à travers le service `translator` (*Translator*[4]). Pour traduire un bloc de texte (appelé un *message*), utilisez la méthode *`trans()`*[5]. Supposons, par exemple, que vous traduisez un simple message dans un contrôleur :

Listing 15-3
```php
public function indexAction()
{
    $t = $this->get('translator')->trans('Symfony2 is great');

    return new Response($t);
}
```

Quand ce code est exécuté, Symfony2 va essayer de traduire le message « Symfony2 is great » en se basant sur la `locale` de l'utilisateur. Pour que cela marche, vous devez dire à Symfony2 comment traduire le message via une « ressource de traduction », qui est une collection de traductions de messages pour une locale donnée. Ce « dictionnaire » de traduction peut être créé en plusieurs formats différents, XLIFF étant le format recommandé :

Listing 15-4
```xml
<!-- messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony2 is great</source>
                <target>J'aime Symfony2</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

4. http://api.symfony.com/master/Symfony/Component/Translation/Translator.html
5. http://api.symfony.com/master/Symfony/Component/Translation/Translator.html#trans()

Maintenant, si la langue de la locale de l'utilisateur est le français, (par ex. `fr_FR` ou `fr_BE`), le message va être traduit par `J'aime Symfony2`.

Le processus de traduction

Pour traduire le message, Symfony2 utilise un processus simple :

- La `locale` de l'utilisateur actuel, qui est stockée dans la requête (ou stockée comme `_locale` en session), est déterminée ;
- Un catalogue des messages traduits est chargé depuis les ressources de traduction définies pour la `locale` (par ex. `fr_FR`). Les messages de la locale de secours sont aussi chargés et ajoutés au catalogue s'ils n'existent pas déjà. Le résultat final est un large « dictionnaire » de traductions. Voir Catalogues de Message pour plus de détails ;
- Si le message est dans le catalogue, la traduction est retournée. Sinon, le traducteur retourne le message original.

Lorsque vous utilisez la méthode `trans()`, Symfony2 cherche la chaîne de caractères exacte à l'intérieur du catalogue de messages approprié et la retourne (si elle existe).

Message avec paramètres de substitution

Parfois, un message contenant une variable a besoin d'être traduit :

Listing 15-5
```php
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello '.$name);

    return new Response($t);
}
```

Cependant, créer une traduction pour cette chaîne de caractères est impossible étant donné que le traducteur va essayer de trouver le message exact, incluant les portions de la variable (par ex. « Hello Ryan » ou « Hello Fabien »). Au lieu d'écrire une traduction pour toutes les itérations possibles de la variable `$name`, vous pouvez remplacer la variable avec un paramètre de substitution (« placeholder ») :

Listing 15-6
```php
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello %name%', array('%name%' => $name));

    new Response($t);
}
```

Symfony2 va maintenant chercher une traduction du message brut (`Hello %name%`) et *ensuite* remplacer les paramètres de substitution avec leurs valeurs. Créer une traduction se fait comme précédemment :

Listing 15-7
```xml
<!-- messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Hello %name%</source>
                <target>Bonjour %name%</target>
            </trans-unit>
        </body>
```

```
11      </file>
12  </xliff>
```

Les paramètres de substitution peuvent prendre n'importe quelle forme puisque le message en entier est reconstruit en utilisant la *fonction strtr*[6] de PHP . Cependant, la notation `%var%` est requise pour les traductions dans les templates Twig, et c'est une convention générale à suivre.

Comme vous l'avez vu, créer une traduction est un processus en deux étapes :

1. Faire abstraction du message qui a besoin d'être traduit en le passant à travers le `Translator`.
2. Créer une traduction pour le message dans chaque locale que vous avez choisi de supporter.

La deuxième étape est faite en créant des catalogues de messages qui définissent les traductions pour chacune des différentes locales.

Catalogues de Message

Lorsqu'un message est traduit, Symfony2 compile un catalogue de messages pour la locale de l'utilisateur et cherche dedans une traduction du message. Un catalogue de messages est comme un dictionnaire de traductions pour une locale spécifique. Par exemple, le catalogue de la locale `fr_FR` pourrait contenir la traduction suivante :

Symfony2 is Great => J'aime Symfony2

C'est la responsabilité du développeur (ou du traducteur) d'une application internationalisée de créer ces traductions. Les traductions sont stockées sur le système de fichiers et reconnues par Symfony, grâce à certaines conventions.

Chaque fois que vous créez une *nouvelle* ressource de traduction (ou installez un bundle qui comprend une ressource de traduction), assurez-vous de vider votre cache afin que Symfony puisse reconnaître les nouvelles traductions :

Listing 15-8
```
1  $ php app/console cache:clear
```

Emplacements des Traductions et Conventions de Nommage

Symfony2 cherche les fichiers de messages (c-a-d les traductions) aux endroits suivants :

- Le répertoire `<répertoire racine du noyau>/Resources/translations`;
- Le répertoire `<répertoire racine du noyau>/Resources/<nom du bundle>/translations`;
- Le répertoire `Resources/translations/` du bundle.

Les répertoires sont listés par ordre de priorité. Cela signifie que vous pouvez surcharger les messages de traduction d'un bundle dans l'un des deux premiers répertoires.

Le système de surcharge se base sur les clés : seules les clés surchargées ont besoin d'être listées dans un fichier de plus grande priorité. Quand une clé n'est pas trouvée dans un fichier de traductions, le service Translator cherchera automatiquement dans les fichiers de moindre priorité.

6. `http://www.php.net/manual/fr/function.strtr.php`

Le nom des fichiers de traductions est aussi important puisque Symfony2 utilise une convention pour déterminer les détails à propos des traductions. Chaque fichier de messages doit être nommé selon le schéma suivant : `domaine.locale.format` :

- **domaine**: Une façon optionnelle d'organiser les messages dans des groupes (par ex. `admin`, `navigation` ou `messages` par défaut) - voir Utiliser les Domaines de Message;
- **locale**: La locale de la traduction (par ex. `en_GB`, `en`, etc);
- **format**: Comment Symfony2 doit charger et analyser le fichier (par ex. `xliff`, `php` ou `yml`).

La valeur du format peut être le nom de n'importe quel format enregistré. Par défaut, Symfony fournit les formats suivants :

- `xliff` : fichier XLIFF ;
- `php` : fichier PHP ;
- `yml` : fichier YAML.

Le choix du format à utiliser dépend de vous, c'est une question de goût.

Vous pouvez également stocker des traductions dans une base de données, ou tout autre système de stockage en fournissant une classe personnalisée implémentant l'interface *`LoaderInterface`*[7].

Créer les Traductions

Le fait de créer des fichiers de traduction est une partie importante de la « localisation » (souvent abrégée *L10n*[8]). Les fichiers de traduction consistent en une série de paires id-traduction pour un domaine et une locale donnés. La source est l'identifiant de la traduction individuelle, et peut être le message dans la locale principale (par exemple « Symfony is great ») de votre application ou un identificateur unique (par exemple « symfony2.great » - voir l'encadré ci-dessous) :

Listing 15-9
```xml
<!-- src/Acme/DemoBundle/Resources/translations/messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony2 is great</source>
                <target>J'aime Symfony2</target>
            </trans-unit>
            <trans-unit id="2">
                <source>symfony2.great</source>
                <target>J'aime Symfony2</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Symfony2 va reconnaître ces fichiers et les utiliser lors de la traduction de « Symfony2 is great » ou de « symfony2.great » dans une locale de langue française (par ex. `fr_FR` or `fr_BE`).

7. http://api.symfony.com/master/Symfony/Component/Translation/Loader/LoaderInterface.html
8. http://fr.wikipedia.org/wiki/Localisation_(informatique)

Utiliser des mots-clés ou des messages

Cet exemple illustre les deux philosophies différentes lors de la création des messages à traduire :

Listing 15-10
```php
$t = $translator->trans('Symfony2 is great');

$t = $translator->trans('symfony2.great');
```

Dans la première méthode, les messages sont écrits dans la langue de la locale par défaut (anglais dans ce cas). Ce message est ensuite utilisé comme « id » lors de la création des traductions.

Dans la seconde méthode, les messages sont en fait des « mots-clés » qui évoquent l'idée du message. Le message mot-clé est ensuite utilisé comme « id » pour toutes les traductions. Dans ce cas, les traductions doivent (aussi) être faites pour la locale par défaut (i.e. pour traduire `symfony2.great` en `Symfony2 is great`).

La deuxième méthode est très pratique, car la clé du message n'aura pas besoin d'être modifiée dans chaque fichier de traduction si vous décidez que le message devrait en fait être « Symfony2 is really great » dans la locale par défaut.

Le choix de la méthode à utiliser dépend entièrement de vous, mais le format « mot-clé » est souvent recommandé.

En outre, les formats de fichiers `php` et `yaml` prennent en charge les ids imbriqués pour éviter de vous répéter, si vous utilisez des mots-clés plutôt que du texte brut comme id :

Listing 15-11
```yaml
symfony2:
    is:
        great: Symfony2 is great
        amazing: Symfony2 is amazing
    has:
        bundles: Symfony2 has bundles
user:
    login: Login
```

Les multiples niveaux sont convertis en paires uniques id / traduction par l'ajout d'un point (.) entre chaque niveau ; donc les exemples ci-dessus sont équivalents à ce qui suit :

Listing 15-12
```yaml
symfony2.is.great: Symfony2 is great
symfony2.is.amazing: Symfony2 is amazing
symfony2.has.bundles: Symfony2 has bundles
user.login: Login
```

Utiliser les Domaines de Message

Comme vous l'avez vu, les fichiers de messages sont organisés par les différentes locales qu'ils traduisent. Pour plus de structure, les fichiers de messages peuvent également être organisés en « domaines ». Lors de la création des fichiers de messages, le domaine est la première partie du nom du fichier. Le domaine par défaut est `messages`. Par exemple, supposons que, par souci d'organisation, les traductions ont été divisées en trois domaines différents : `messages`, `admin` et `navigation`. La traduction française aurait les fichiers de message suivants :

- `messages.fr.xliff`
- `admin.fr.xliff`
- `navigation.fr.xliff`

Lors de la traduction de chaînes de caractères qui ne sont pas dans le domaine par défaut (`messages`), vous devez spécifier le domaine comme troisième argument de `trans()` :

Listing 15-13
```php
$this->get('translator')->trans('Symfony2 is great', array(), 'admin');
```

Symfony2 va maintenant chercher le message dans le domaine `admin` de la locale de l'utilisateur.

Gérer la Locale de l'Utilisateur

La locale de l'utilisateur courant est stockée dans la requête et est accessible via l'objet `request` :

Listing 15-14
```php
// Accéder à l'objet Request dans un contrôleur standard
$request = $this->getRequest();

$locale = $request->getLocale();

$request->setLocale('en_US');
```

Il est aussi possible de stocker la locale en session plutôt qu'en requête. Si vous faites cela, toutes les requêtes auront la même locale.

Listing 15-15
```php
$this->get('session')->set('_locale', 'en_US');
```

Lisez le chapitre *La locale et l'URL* pour voir comment définir la locale dans vos routes.

Solution de Secours et Locale par Défaut

Si la locale n'a pas été explicitement définie dans la session, le paramètre de configuration `fallback_locale` va être utilisé par le `Translator`. Le paramètre est défini comme `en` par défaut (voir Configuration).

Alternativement, vous pouvez garantir que la locale soit définie dans chaque requête de l'utilisateur en définissant le paramètre `default_locale` du framework :

Listing 15-16
```yaml
# app/config/config.yml
framework:
    default_locale: en
```

New in version 2.1: Le paramètre `default_locale` était à la base défini dans la clé session, cependant cela a changé dans la version 2.1. C'est parce que la locale est maintenant définie dans la requête et non plus dans la session

La locale et l'URL

Puisque vous pouvez stocker la locale de l'utilisateur dans la session, il peut être tentant d'utiliser la même URL pour afficher une ressource dans de nombreuses langues différentes en se basant sur la locale de l'utilisateur. Par exemple, `http://www.example.com/contact` pourrait afficher le contenu en anglais pour un utilisateur, et en français pour un autre utilisateur. Malheureusement, cela viole une règle fondamentale du Web qui dit qu'une URL particulière retourne la même ressource indépendamment de l'utilisateur. Pour enfoncer encore plus le clou, quelle version du contenu serait indexée par les moteurs de recherche ?

Une meilleure politique est d'inclure la locale dans l'URL. Ceci est entièrement pris en charge par le système de routage en utilisant le paramètre spécial `_locale` :

Listing 15-17
```yaml
contact:
    pattern:   /{_locale}/contact
    defaults:  { _controller: AcmeDemoBundle:Contact:index, _locale: en }
    requirements:
        _locale: en|fr|de
```

Lorsque vous utilisez le paramètre spécial *_locale* dans une route, la locale correspondante sera *automatiquement définie dans la session de l'utilisateur*. En d'autres termes, si un utilisateur visite l'URI `/fr/contact`, la locale `fr` sera automatiquement définie comme la locale de sa session.

Vous pouvez maintenant utiliser la locale de l'utilisateur pour créer des routes pointant vers d'autres pages traduites de votre application.

Pluralisation

La pluralisation des messages est un sujet difficile, car les règles peuvent être assez complexes. Par exemple, voici la représentation mathématique des règles de la pluralisation russe :

Listing 15-18
```
(($number % 10 == 1) && ($number % 100 != 11)) ? 0 : ((($number % 10 >= 2) && ($number % 10
<= 4) && (($number % 100 < 10) || ($number % 100 >= 20))) ? 1 : 2);
```

Comme vous pouvez le voir, en russe, vous pouvez avoir trois différentes formes de pluriel, chacune donnant un index de 0, 1 ou 2. Pour chaque forme, le pluriel est différent, et ainsi la traduction est également différente.

Quand une traduction a des formes différentes dues à la pluralisation, vous pouvez fournir toutes les formes comme une chaîne de caractères séparée par un pipe (`|`):

Listing 15-19
```
'There is one apple|There are %count% apples'
```

Pour traduire des messages pluralisés, utilisez la méthode *`transChoice()`*[9] :

Listing 15-20
```php
$t = $this->get('translator')->transChoice(
    'There is one apple|There are %count% apples',
    10,
    array('%count%' => 10)
);
```

Le second paramètre (`10` dans cet exemple), est le *nombre* d'objets étant décrits et est utilisé pour déterminer quelle traduction utiliser et aussi pour définir le paramètre de substitution `%count%`.

En se basant sur le nombre donné, le traducteur choisit la bonne forme du pluriel. En anglais, la plupart des mots ont une forme singulière quand il y a exactement un objet et un pluriel pour tous les autres nombres (0, 2, 3 ...). Ainsi, si `count` vaut `1`, le traducteur va utiliser la première chaîne de caractères (`There is one apple`) comme traduction. Sinon, il va utiliser `There are %count% apples`.

Voici la traduction française :

Listing 15-21
```
'Il y a %count% pomme|Il y a %count% pommes'
```

9. http://api.symfony.com/master/Symfony/Component/Translation/Translator.html#transChoice()

Même si la chaîne de caractères se ressemble (elle est constituée de deux sous-chaînes séparées par un pipe), les règles françaises sont différentes : la première forme (pas de pluriel) est utilisée lorsque `count` vaut `0` ou `1`. Ainsi, le traducteur va utiliser automatiquement la première chaîne (`Il y a %count% pomme`) lorsque `count` vaut `0` ou `1`.

Chaque locale a son propre ensemble de règles, certaines ayant jusqu'à six différentes formes plurielles avec des règles complexes pour déterminer quel nombre correspond à quelle forme du pluriel. Les règles sont assez simples pour l'anglais et le français, mais pour le russe, vous auriez voulu un indice pour savoir quelle règle correspond à quelle chaîne de caractères. Pour aider les traducteurs, vous pouvez éventuellement « taguer » chaque chaîne :

Listing 15-22
```
'one: There is one apple|some: There are %count% apples'

'none_or_one: Il y a %count% pomme|some: Il y a %count% pommes'
```

Les tags sont seulement des indices pour les traducteurs et n'affectent pas la logique utilisée pour déterminer quelle forme de pluriel utiliser. Les tags peuvent être toute chaîne descriptive qui se termine par un deux-points (`:`). Les tags n'ont pas besoin d'être les mêmes dans le message original que dans la traduction.

Comme les tags sont optionnels, le traducteur ne les utilise pas (il va seulement obtenir une chaîne de caractères en fonction de sa position dans la chaîne).

Intervalle Explicite de Pluralisation

La meilleure façon de pluraliser un message est de laisser Symfony2 utiliser sa logique interne pour choisir quelle chaîne utiliser en se basant sur un nombre donné. Parfois, vous aurez besoin de plus de contrôle ou vous voudrez une traduction différente pour des cas spécifiques (pour `0`, ou lorsque le nombre est négatif, par exemple). Pour de tels cas, vous pouvez utiliser des intervalles mathématiques explicites :

Listing 15-23
```
'{0} There are no apples|{1} There is one apple|]1,19] There are %count% apples|[20,Inf]
There are many apples'
```

Les intervalles suivent la notation *ISO 31-11*[10] . La chaîne de caractères ci-dessus spécifie quatre intervalles différents : exactement `0`, exactement `1`, `2-19`, et `20` et plus.

Vous pouvez également mélanger les règles mathématiques explicites et les règles standards. Dans ce cas, si le nombre ne correspond pas à un intervalle spécifique, les règles standards prennent effet après la suppression des règles explicites :

Listing 15-24
```
'{0} There are no apples|[20,Inf] There are many apples|There is one apple|a_few: There are
%count% apples'
```

Par exemple, pour `1` pomme (« apple »), la règle standard `There is one apple` va être utilisée. Pour `2-19` pommes (« apples »), la seconde règle standard `There are %count% apples` va être sélectionnée.

Une classe *`Interval`*[11] peut représenter un ensemble fini de nombres :

Listing 15-25
```
{1,2,3,4}
```

Ou des nombres entre deux autres nombres :

10. http://en.wikipedia.org/wiki/ISO_31-11
11. http://api.symfony.com/master/Symfony/Component/Translation/Interval.html

Listing 15-26
```
[1, +Inf[
]-1,2[
```

Le délimiteur gauche peut-être `[` (inclusif) ou `]` (exclusif). Le delimiteur droit peut-être `[` (exclusif) ou `]` (inclusif). En sus des nombres, vous pouvez utiliser `-Inf` and `+Inf` pour l'infini.

Traductions dans les Templates

La plupart du temps, les traductions surviennent dans les templates. Symfony2 supporte nativement les deux types de templates que sont Twig et PHP.

Templates Twig

Symfony2 fournit des balises Twig spécialisées (`trans` et `transchoice`) pour aider à la traduction des *blocs statiques de texte* :

Listing 15-27
```
{% trans %}Hello %name%{% endtrans %}

{% transchoice count %}
    {0} There is no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

La balise `transchoice` prend automatiquement la variable `%count%` depuis le contexte actuel et la passe au traducteur. Ce mécanisme fonctionne seulement lorsque vous utilisez un paramètre de substitution suivi du pattern `%var%`.

Si vous avez besoin d'utiliser le caractère pourcentage (%) dans une chaîne de caractères, échappez-le en le doublant : `{% trans %}Percent: %percent%%%{% endtrans %}`

Vous pouvez également spécifier le domaine de messages et passer quelques variables supplémentaires :

Listing 15-28
```
{% trans with {'%name%': 'Fabien'} from "app" %}Hello %name%{% endtrans %}

{% trans with {'%name%': 'Fabien'} from "app" into "fr" %}Hello %name%{% endtrans %}

{% transchoice count with {'%name%': 'Fabien'} from "app" %}
    {0} There is no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

Les filtres `trans` et `transchoice` peuvent être utilisés pour traduire les *textes de variable* ainsi que les expressions complexes :

Listing 15-29
```
{{ message|trans }}

{{ message|transchoice(5) }}

{{ message|trans({'%name%': 'Fabien'}, "app") }}

{{ message|transchoice(5, {'%name%': 'Fabien'}, 'app') }}
```

Utiliser les balises ou filtres de traduction a le même effet, mais avec une différence subtile : l'échappement automatique en sortie est appliqué uniquement aux variables traduites via un filtre. En d'autres termes, si vous avez besoin d'être sûr que votre variable traduite n'est *pas* échappée en sortie, vous devez appliquer le filtre brut après le filtre de traduction :

Listing 15-30
```twig
{# le texte traduit entre les balises n'est jamais échappé #}
{% trans %}
    <h3>foo</h3>
{% endtrans %}

{% set message = '<h3>foo</h3>' %}

{# une variable traduite via un filtre est échappée par défaut #}
{{ message|trans|raw }}

{# mais les chaînes de caractères statiques ne sont jamais échappées #}
{{ '<h3>foo</h3>'|trans }}
```

New in version 2.1: Vous pouvez maintenant définir un domaine de traduction pour un template Twig entier avec une seule balise :

Listing 15-31
```twig
{% trans_default_domain "app" %}
```

Notez que cela n'affecte que le template courant, pas les templates inclus pour éviter les effets de bord.

Templates PHP

Le service de traduction est accessible dans les templates PHP à travers l'outil d'aide `translator` :

Listing 15-32
```php
<?php echo $view['translator']->trans('Symfony2 is great') ?>

<?php echo $view['translator']->transChoice(
    '{0} There is no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10)
) ?>
```

Forcer la Locale du Traducteur

Lors de la traduction d'un message, Symfony2 utilise la locale de la requête courante ou la locale de secours (« fallback locale ») si nécessaire. Vous pouvez également spécifier manuellement la locale à utiliser pour la traduction :

Listing 15-33
```php
$this->get('translator')->trans(
    'Symfony2 is great',
    array(),
    'messages',
    'fr_FR'
```

```
);

$this->get('translator')->transChoice(
    '{0} There is no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10),
    'messages',
    'fr_FR'
);
```

Traduire le Contenu d'une Base de Données

La traduction du contenu de bases de données devrait être traitée par Doctrine grâce à l'extension *Translatable Extension*[12]. Pour plus d'informations, voir la documentation pour cette bibliothèque.

Traduire les messages de contraintes

La meilleure manière de comprendre les contraintes de traduction est de les voir en action. Pour commencer, supposez que vous avez créé un objet PHP que vous avez besoin d'utiliser quelque part dans votre application :

Listing 15-34
```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    public $name;
}
```

Ajoutez des contraintes avec l'une des méthodes supportées. Définissez le texte source à traduire dans l'option message. Par exemple, pour garantir qu'une propriété $name n'est pas vide, ajoutez le code suivant :

Listing 15-35
```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        name:
            - NotBlank: { message: "author.name.not_blank" }
```

Créez un fichier de traduction pour le catalogue `validators` pour les messages de contraintes, typiquement dans le répertoire `Resources/translations/` du bundle. Lisez Catalogues de Message pour en savoir plus

Listing 15-36
```
<!-- validators.en.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>author.name.not_blank</source>
```

12. https://github.com/l3pp4rd/DoctrineExtensions

```
                <target>Saisissez un nom</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Résumé

Avec le composant Traduction de Symfony2, créer une application internationalisée n'a plus besoin d'être un processus douloureux et se résume simplement à quelques étapes basiques :

- Extraire les messages dans votre application en entourant chacun d'entre eux par la méthode *`trans()`*[13] ou par la méthode *`transChoice()`*[14];
- Traduire chaque message dans de multiples locales en créant des fichiers de message de traduction. Symfony2 découvre et traite chaque fichier grâce à leur nom qui suit une convention spécifique ;
- Gérer la locale de l'utilisateur, qui est stockée dans la requête, ou une fois pour toutes en session.

13. http://api.symfony.com/master/Symfony/Component/Translation/Translator.html#trans()
14. http://api.symfony.com/master/Symfony/Component/Translation/Translator.html#transChoice()

Chapter 16
Service Container

Une application PHP moderne est plein d'objets. Un objet peut faciliter l'envoi des messages e-mail, tandis qu'un autre peut vous permettre de persister les informations dans une base de données. Dans votre application, vous pouvez créer un objet qui gère votre inventaire de produits, ou tout autre objet qui traite des données via une API tierce. Le fait est qu'une application moderne fait beaucoup de choses et est organisée entre de nombreux objets qui gèrent chaque tâche.

Ce chapitre traite d'un objet spécial PHP dans Symfony2 qui vous aide à instancier, organiser et récupérer les nombreux objets de votre application. Cet objet, appelé un conteneur de services, vous permettra de standardiser et centraliser la façon dont les objets sont construits dans votre application. Le conteneur vous facilite la vie, est super rapide, et met en valeur une architecture qui encourage un code réutilisable et découplé. Puisque toutes les classes fondamentales de Symfony2 utilisent le conteneur, vous allez apprendre comment étendre, configurer et utiliser n'importe quel objet dans Symfony2. En bien des aspects, le conteneur de services est le principal responsable de la vitesse et de l'extensibilité de Symfony2.

Enfin, configurer et utiliser le conteneur de services est facile. A la fin de ce chapitre, vous serez à l'aise pour créer vos propres objets via le conteneur et pour personnaliser des objets provenant de bundles tiers. Vous allez commencer à écrire du code qui est plus réutilisable, testable et découplé, tout simplement parce le conteneur de services facilite l'écriture de code de qualité.

Qu'est-ce qu'un Service ?

Plus simplement, un *Service* désigne tout objet PHP qui effectue une sorte de tâche « globale ». C'est un nom générique utilisé en informatique pour décrire un objet qui est créé dans un but précis (par ex. l'envoi des emails). Chaque service est utilisé tout au long de votre application lorsque vous avez besoin de la fonctionnalité spécifique qu'il fournit. Vous n'avez pas à faire quelque chose de spécial pour fabriquer un service : il suffit d'écrire une classe PHP avec un code qui accomplit une tâche spécifique. Félicitations, vous venez tout juste de créer un service !

En règle générale, un objet PHP est un service s'il est utilisé de façon globale dans votre application. Un seul service `Mailer` est utilisé globalement pour envoyer des messages email tandis que les nombreux objets `Message` qu'il délivre ne sont *pas* des services. De même, un objet `Product` n'est pas un service, mais un objet qui persiste des objets `Product` dans une base de données *est* un service.

Alors quel est l'avantage ? L'avantage de réfléchir sur les « services » est que vous commencez à penser à séparer chaque morceau de fonctionnalité dans votre application dans une série de services. Puisque chaque service ne réalise qu'une fonction, vous pouvez facilement accéder à chaque service et utiliser ses fonctionnalités chaque fois que vous en avez besoin. Chaque service peut également être plus facilement testé et configuré puisqu'il est séparé des autres fonctionnalités de votre application. Cette idée est appelée *service-oriented architecture*[1] et n'est pas unique à Symfony2 ou encore PHP. Structurer votre application autour d'un ensemble de classes de services indépendants est une bonne pratique orientée objet célèbre et fiable. Ces compétences sont les clés pour devenir un bon développeur dans presque tous les langages.

Définition d'un Conteneur de Services

Un *Conteneur de services* (« service container » ou « *dependency injection container* » en anglais) est simplement un objet PHP qui gère l'instanciation des services (c-a-d objets).

Par exemple, supposons que vous ayez une simple classe PHP qui envoie des messages email. Sans un conteneur de services, vous devez manuellement créer l'objet chaque fois que vous en avez besoin :

Listing 16-1
```php
use Acme\HelloBundle\Mailer;

$mailer = new Mailer('sendmail');
$mailer->send('ryan@foobar.net', ... );
```

Ceci est assez facile. La classe imaginaire `Mailer` vous permet de configurer la méthode utilisée pour envoyer les messages par e-mail (par exemple `sendmail`, `smtp`, etc) Mais que faire si vous voulez utiliser le service mailer ailleurs ? Vous ne voulez certainement pas répéter la configuration du mailer *chaque* fois que vous devez utiliser l'objet `Mailer`. Que se passe-t-il si vous avez besoin de changer le `transport` de `sendmail` à `smtp` partout dans l'application ? Vous auriez besoin de chercher chaque endroit où vous avez créé un service `Mailer` et de le changer.

Créer/Configurer les services dans le Conteneur

Une meilleure solution est de laisser le conteneur de services créer l'objet `Mailer` pour vous. Pour que cela fonctionne, vous devez *spécifier* au conteneur comment créer le `Mailer`. Cela se fait via la configuration, qui peut être spécifiée en YAML, XML ou PHP :

Listing 16-2
```yaml
# app/config/config.yml
services:
    my_mailer:
        class:        Acme\HelloBundle\Mailer
        arguments:    [sendmail]
```

1. http://wikipedia.org/wiki/Service-oriented_architecture

Lorsque Symfony2 s'initialise, il construit le conteneur de services en utilisant la configuration de l'application (`app/config/config.yml` par défaut). Le fichier exact qui est chargé est dicté par la méthode `AppKernel::registerContainerConfiguration()`, qui charge un fichier de configuration spécifique à l'environnement (par exemple `config_dev.yml` pour l'environnement de `dev` ou `config_prod.yml` pour la `prod`).

Une instance de l'objet `Acme\HelloBundle\Mailer` est maintenant disponible via le conteneur de services. Le conteneur est disponible dans tous les contrôleurs traditionnels de Symfony2 où vous pouvez accéder aux services du conteneur via la méthode de raccourci `get()` :

Listing 16-3
```php
class HelloController extends Controller
{
    // ...

    public function sendEmailAction()
    {
        // ...
        $mailer = $this->get('my_mailer');
        $mailer->send('ryan@foobar.net', ... );
    }
}
```

Lorsque vous demandez le service `my_mailer` du conteneur, le conteneur construit l'objet et le retourne. Ceci est un autre avantage majeur d'utiliser le conteneur de services. A savoir, un service n'est *jamais* construit avant qu'il ne soit nécessaire. Si vous définissez un service et ne l'utilisez jamais sur une demande, le service n'est jamais créé. Cela permet d'économiser la mémoire et d'augmenter la vitesse de votre application. Cela signifie aussi qu'il y a très peu ou pas d'impact de performance en définissant beaucoup de services. Les services qui ne sont jamais utilisés ne sont jamais construits.

Comme bonus supplémentaire, le service `Mailer` est seulement créé une fois et la même instance est retournée chaque fois que vous demandez le service. Ceci est presque toujours le comportement dont vous aurez besoin (c'est plus souple et plus puissant), mais vous apprendrez comment configurer un service qui a de multiples instances dans l'article du Cookbook « *Comment travailler avec les champs d'applications (« scopes » en anglais)* ».

Paramètres de Service

La création de nouveaux services (c-a-d objets) via le conteneur est assez simple. Les paramètres rendent les définitions de services plus organisées et flexibles :

Listing 16-4
```yaml
# app/config/config.yml
parameters:
    my_mailer.class:      Acme\HelloBundle\Mailer
    my_mailer.transport:  sendmail

services:
    my_mailer:
        class:        "%my_mailer.class%"
        arguments:    [%my_mailer.transport%]
```

Le résultat final est exactement le même que précédemment - la différence est seulement dans la *manière* dont vous avez défini le service. En entourant les chaînes `my_mailer.class` et `my_mailer.transport` par le signe pour cent (`%`), le conteneur sait qu'il faut chercher des paramètres avec ces noms. Quand le conteneur est construit, il cherche la valeur de chaque paramètre et l'utilise dans la définition du service.

Le signe pour cent au sein d'un paramètre ou d'un argument, et qui fait partie de la chaîne de caractères, doit être échappé par un autre signe pour cent :

Listing 16-5
```
<argument type="string">http://symfony.com/?foo=%%s&bar=%%d</argument>
```

Le but des paramètres est de fournir l'information dans les services. Bien sûr, il n'y avait rien de mal à définir le service sans utiliser de paramètre. Les paramètres, cependant, ont plusieurs avantages :

- la séparation et l'organisation de toutes les « options » de service sous une seule clé de `paramètres` ;
- les valeurs de paramètres peuvent être utilisées dans de multiples définitions de service ;
- Lors de la création d'un service dans un bundle (vous allez voir ceci sous peu), utiliser les paramètres permet au service d'être facilement personnalisé dans votre application.

Le choix d'utiliser ou non des paramètres dépend de vous. Les bundles tiers de haute qualité utiliseront *toujours* les paramètres puisqu'ils rendent le service stocké dans le conteneur plus configurable. Pour les services dans votre application, cependant, vous pouvez ne pas avoir besoin de la flexibilité des paramètres.

Tableaux de paramètres

Les paramètres ne sont pas obligatoirement des chaînes de caractères, ils peuvent aussi être des tableaux. Pour le format XML, vous devez utiliser l'attribut type="collection" pour tous les paramètres qui sont des tableaux.

Listing 16-6
```yaml
# app/config/config.yml
parameters:
    my_mailer.gateways:
        - mail1
        - mail2
        - mail3
    my_multilang.language_fallback:
        en:
            - en
            - fr
        fr:
            - fr
            - en
```

Importer d'autres Ressources de Configuration de Conteneur

Dans cette section, vous allez faire référence aux fichiers de configuration de service comme des *ressources*. C'est pour souligner le fait que, alors que la plupart des ressources de configuration sont des fichiers (par exemple YAML, XML, PHP), Symfony2 est si flexible que la configuration pourrait être chargée de n'importe où (par exemple une base de données ou même via un service web externe).

Le conteneur de services est construit en utilisant une ressource de configuration unique (`app/config/config.yml` par défaut). Toutes les autres configurations de service (y compris la configuration du noyau de Symfony2 et des bundle tiers) doivent être importées à l'intérieur de ce fichier d'une manière ou d'une autre. Cela vous donne une flexibilité absolue sur les services dans votre application.

La configuration des services externes peut être importée de deux manières différentes. La première, et la plus commune, consiste à utiliser la directive `imports`. Plus tard, vous apprendrez la seconde méthode qui est la méthode flexible et préférée pour l'importation de configuration de services des bundles tiers.

Importer la Configuration avec `imports`

Jusqu'ici, vous avez placé notre définition de conteneur de service `my_mailer` directement dans le fichier de configuration de l'application (par exemple `app/config/config.yml`). Bien sûr, puisque la classe `Mailer` elle-même vit à l'intérieur de `AcmeHelloBundle`, il est plus logique de mettre la définition du conteneur `my_mailer` à l'intérieur du bundle aussi.

Tout d'abord, déplacez la définition du conteneur `my_mailer` dans un nouveau fichier de ressource de conteneur à l'intérieur d' `AcmeHelloBundle`. Si les répertoires `Resources` ou `Resources/config` n'existent pas, créez-les.

Listing 16-7
```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    my_mailer.class:      Acme\HelloBundle\Mailer
    my_mailer.transport:  sendmail

services:
    my_mailer:
        class:        "%my_mailer.class%"
        arguments:    [%my_mailer.transport%]
```

La définition elle-même n'a pas changé, seulement son emplacement. Bien sûr, le conteneur de service ne connaît pas le nouveau fichier de ressources. Heureusement, vous pouvez facilement importer le fichier de ressources en utilisant la clé `imports` dans la configuration de l'application.

Listing 16-8
```
# app/config/config.yml
imports:
    hello_bundle:
         - { resource: @AcmeHelloBundle/Resources/config/services.yml }
```

La directive `imports` permet à votre application d'inclure des ressources de configuration de conteneur de services de n'importe quel autre emplacement (le plus souvent à partir de bundles). L'emplacement `resource`, pour les fichiers, est le chemin absolu du fichier de ressource. La syntaxe spéciale `@AcmeHello` résout le chemin du répertoire du bundle `AcmeHelloBundle`. Cela vous aide à spécifier le chemin vers la ressource sans se soucier plus tard, si vous déplacez le `AcmeHelloBundle` dans un autre répertoire.

Importer la Configuration via les Extensions de Conteneur

Quand vous développerez avec Symfony2, vous utiliserez le plus souvent la directive `imports` pour importer la configuration du conteneur des bundles que vous avez créé spécifiquement pour votre application. Les configurations des conteneurs des bundles tiers, y compris les services du noyau de Symfony2, sont habituellement chargées en utilisant une autre méthode qui est plus souple et facile à configurer dans votre application.

Voici comment cela fonctionne. En interne, chaque bundle définit ses services comme vous l'avez vu jusqu'à présent. A savoir, un bundle utilise un ou plusieurs fichiers de ressources de configuration (généralement XML) pour spécifier les paramètres et les services pour ce bundle. Cependant, au lieu d'importer chacune de ces ressources directement à partir de la configuration de votre application en utilisant la directive `imports`, vous pouvez simplement invoquer une *extension du conteneur de services* à l'intérieur du bundle qui fait le travail pour vous. Une extension de conteneur de services est une classe PHP créée par l'auteur du bundle afin d'accomplir deux choses :

- importer toutes les ressources du conteneur de services nécessaires pour configurer les services pour le bundle ;

- fournir une configuration sémantique, simple de sorte que le bundle peut être configuré sans interagir avec les paramètres de la configuration du conteneur de services du bundle.

En d'autres termes, une extension de conteneur de services configure les services pour un bundle en votre nom. Et comme vous le verrez dans un instant, l'extension fournit une interface pratique, de haut niveau pour configurer le bundle.

Prenez le `FrameworkBundle` - le bundle noyau du framework Symfony2 - comme un exemple. La présence du code suivant dans votre configuration de l'application invoque l'extension du conteneur de services à l'intérieur du `FrameworkBundle` :

Listing 16-9
```yaml
# app/config/config.yml
framework:
    secret:          xxxxxxxxxx
    form:            true
    csrf_protection: true
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
    # ...
```

Lorsque la configuration est analysée, le conteneur cherche une extension qui peut gérer la directive de configuration du `framework`. L'extension en question, qui vit dans le `FrameworkBundle`, est invoquée et la configuration du service pour le `FrameworkBundle` est chargée. Si vous retirez la clé `framework` de votre fichier de configuration de l'application entièrement, les services noyau de Symfony2 ne seront pas chargés. Le fait est que vous avez la maîtrise : le framework Symfony2 ne contient pas de magie et n'effectue aucune action dont vous n'avez pas le contrôle dessus.

Bien sûr, vous pouvez faire beaucoup plus que simplement « activer » l'extension du conteneur de services du `FrameworkBundle`. Chaque extension vous permet de facilement personnaliser le bundle, sans vous soucier de la manière dont les services internes sont définis.

Dans ce cas, l'extension vous permet de personnaliser le `error_handler`, `csrf_protection`, `router` et bien plus encore. En interne, le `FrameworkBundle` utilise les options spécifiées ici pour définir et configurer les services qui lui sont spécifiques. Le bundle se charge de créer tous les `paramètres` et `services` nécessaires pour le conteneur du service, tout en permettant une grande partie de la configuration d'être facilement personnalisée. Comme bonus supplémentaire, la plupart des extensions du conteneur de services sont assez malines pour effectuer la validation - vous informant des options qui sont manquantes ou du mauvais type de données.

Lors de l'installation ou la configuration d'un bundle, consultez la documentation du bundle pour savoir comment installer et configurer les services pour le bundle. Les options disponibles pour les bundles du noyau peuvent être trouvées à *Reference Guide*.

Nativement, le conteneur de services reconnaît seulement les directives `parameters`, `services`, et `imports`. Toutes les autres directives sont gérées par une extension du conteneur de service.

Si vous voulez exposer une configuration conviviale dans vos propres bundles, lisez l'entrée du cookbook "*Comment exposer une configuration sémantique pour un Bundle*".

Référencer (Injecter) les Services

Jusqu'à présent, notre service originel `my_mailer` est simple : il suffit d'un seul paramètre dans son constructeur, qui est facilement configurable. Comme vous le verrez, la vraie puissance du conteneur est démontrée lorsque vous avez besoin de créer un service qui dépend d'un ou plusieurs autres services dans le conteneur.

Commençons par un exemple. Supposons que vous ayez un nouveau service, `NewsletterManager`, qui aide à gérer la préparation et l'envoi d'un message email à une liste d'adresses. Bien sûr, le service `my_mailer` excelle vraiment pour envoyer des messages email, donc vous allez l'utiliser dans `NewsletterManager` pour gérer l'envoi effectif des messages. Cette fausse classe pourrait ressembler à quelque chose comme ceci :

Listing 16-10
```php
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Sans utiliser le conteneur de services, vous pouvez créer une nouvelle `NewsletterManager` assez facilement à l'intérieur d'un contrôleur :

Listing 16-11
```php
public function sendNewsletterAction()
{
    $mailer = $this->get('my_mailer');
    $newsletter = new Acme\HelloBundle\Newsletter\NewsletterManager($mailer);
    // ...
}
```

Cette approche est pas mal, mais si nous décidons plus tard que la classe `NewsletterManager` a besoin d'un deuxième ou troisième paramètre de constructeur ? Que se passe-t-il si nous décidons de refactoriser notre code et de renommer la classe ? Dans les deux cas, vous auriez besoin de trouver tous les endroits où le `NewsletterManager` a été instancié et de le modifier. Bien sûr, le conteneur de services vous donne une option beaucoup plus attrayante :

Listing 16-12
```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

services:
    my_mailer:
        # ...
    newsletter_manager:
        class:     "%newsletter_manager.class%"
        arguments: [@my_mailer]
```

En YAML, la syntaxe spéciale `@my_mailer` indique au conteneur de chercher un service nommé `my_mailer` et de transmettre cet objet dans le constructeur de `NewsletterManager`. Dans ce cas, cependant, le service spécifié `my_mailer` doit exister. Si ce n'est pas le cas, une exception sera levée. Vous pouvez marquer vos dépendances comme facultatives - nous en parlerons dans la section suivante.

Utiliser des références est un outil très puissant qui vous permet de créer des classes de services indépendantes avec des dépendances bien définies. Dans cet exemple, le service `newsletter_manager`

a besoin du service `my_mailer` afin de fonctionner. Lorsque vous définissez cette dépendance dans le conteneur de service, le conteneur prend soin de tout le travail de l'instanciation des objets.

Dépendances optionnelles : Setter Injection

L'injection de dépendances dans le constructeur de cette manière est un excellent moyen de s'assurer que la dépendance est disponible pour utilisation. Si vous avez des dépendances optionnelles pour une classe, alors la méthode « setter injection » peut être une meilleure option. Cela signifie d'injecter la dépendance en utilisant un appel de méthode plutôt que par le constructeur. La classe devrait ressembler à ceci :

Listing 16-13
```php
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

L'injection de la dépendance par la méthode setter a juste besoin d'un changement de la syntaxe :

Listing 16-14
```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

services:
    my_mailer:
        # ...
    newsletter_manager:
        class:     "%newsletter_manager.class%"
        calls:
            - [ setMailer, [ @my_mailer ] ]
```

Les approches présentées dans cette section sont appelées « constructor injection » et « setter injection« ». Le conteneur de service Symfony2 supporte aussi « property injection« ».

Rendre les Références Optionnelles

Parfois, un de vos services peut avoir une dépendance optionnelle, ce qui signifie que la dépendance n'est pas requise par le service pour fonctionner correctement. Dans l'exemple ci-dessus, le service `my_mailer` *doit* exister, sinon une exception sera levée. En modifiant les définitions du service `newsletter_manager`, vous pouvez rendre cette référence optionnelle. Le conteneur va ensuite l'injecter si elle existe et ne rien faire si ce n'est pas le cas :

Listing 16-15
```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
```

```
    # ...

services:
    newsletter_manager:
        class:     "%newsletter_manager.class%"
        arguments: [@?my_mailer]
```

En YAML, la syntaxe spéciale `@?` indique au conteneur de service que la dépendance est optionnelle. Bien sûr, le `NewsletterManager` doit être aussi écrit pour permettre une dépendance optionnelle :

Listing 16-16
```php
public function __construct(Mailer $mailer = null)
{
    // ...
}
```

Services de Bundle Tiers et Noyau de Symfony

Étant donné que Symfony2 et tous les bundles tiers configurent et récupèrent leurs services via le conteneur, vous pouvez facilement y accéder, ou même les utiliser dans vos propres services. Pour garder les choses simples, par défaut Symfony2 n'exige pas que les contrôleurs soient définis comme des services. Par ailleurs Symfony2 injecte l'ensemble du conteneur de services dans votre contrôleur. Par exemple, pour gérer le stockage des informations sur une session utilisateur, Symfony2 fournit un service `session`, auquel vous pouvez accéder de l'intérieur d'un contrôleur standard comme suit :

Listing 16-17
```php
public function indexAction($bar)
{
    $session = $this->get('session');
    $session->set('foo', $bar);

    // ...
}
```

Dans Symfony2, vous allez constamment utiliser les services fournis par le noyau de Symfony ou autres bundles tiers pour effectuer des tâches telles que rendre des templates (`templating`), envoyer des emails (`mailer`), ou d'accéder à des informations sur la requête (`request`).

Vous pouvez aller plus loin en utilisant ces services à l'intérieur des services que vous avez créés pour votre application. Modifions le `NewsletterManager` afin d'utiliser le vrai service `mailer` de Symfony2 (au lieu du faux `my_mailer`). Passons aussi le service du moteur de template à `NewsletterManager` afin qu'il puisse générer le contenu de l'email via un template :

Listing 16-18
```php
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\Templating\EngineInterface;

class NewsletterManager
{
    protected $mailer;

    protected $templating;

    public function __construct(\Swift_Mailer $mailer, EngineInterface $templating)
    {
        $this->mailer = $mailer;
        $this->templating = $templating;
```

```
15      }
16
17      // ...
18  }
```

Configurer le conteneur de services est facile :

Listing 16-19
```
services:
    newsletter_manager:
        class:     "%newsletter_manager.class%"
        arguments: [@mailer, @templating]
```

Le service `newsletter_manager` a désormais accès aux services noyau `mailer` et `templating`. C'est une façon commune de créer des services spécifiques à votre application qui exploitent la puissance des différents services au sein du framework.

Soyez sûr que l'entrée `swiftmailer` apparaît dans votre configuration de l'application. Comme vous l'avez mentionné dans *Importer la Configuration via les Extensions de Conteneur*, la clé `swiftmailer` invoque l'extension du service de `SwiftmailerBundle`, qui déclare le service `mailer`.

Tags

De la même manière qu'un billet de blog sur le Web pourrait être tagué avec des noms telles que « Symfony » ou « PHP », les services configurés dans votre conteneur peuvent également être tagués. Dans le conteneur de services, un tag laisse supposer que le service est censé être utilisé dans un but précis. Prenons l'exemple suivant :

Listing 16-20
```
services:
    foo.twig.extension:
        class: Acme\HelloBundle\Extension\FooExtension
        tags:
            -  { name: twig.extension }
```

Le tag `twig.extension` est un tag spécial que le `TwigBundle` utilise pendant la configuration. En donnant au service ce tag `twig.extension`, le bundle sait que le service `foo.twig.extension` devrait être enregistré comme une extension Twig avec Twig. En d'autres termes, Twig trouve tous les services taggés avec `twig.extension` et les enregistre automatiquement comme des extensions.

Les tags, alors, sont un moyen de dire aux bundles de Symfony2 ou tiers que votre service doit être enregistré ou utilisé d'une manière spéciale par le bundle.

Ce qui suit est une liste de tags disponibles avec les bundles noyau de Symfony2. Chacun d'eux a un effet différent sur votre service et de nombreux tags nécessitent des paramètres supplémentaires (au-delà du paramètre `name`).

Pour une liste de tous les tags disponibles dans le coeur du Framework Symfony, consultez *Les Tags de l'Injection de Dépendances*.

Débugger les services

Vous pouvez voir quels services sont enregistrés dans le conteneur grâce à la console. Pour afficher tous les services et les classes de chacun d'entre eux, exécutez :

Listing 16-21
```
$ php app/console container:debug
```

Par défaut, seuls les services publics sont affichés, mais vous pouvez également voir les services privés :

Listing 16-22
```
$ php app/console container:debug --show-private
```

Vous pouvez obtenir des informations détaillées sur un service particulier en spécifiant son identifiant :

Listing 16-23
```
$ php app/console container:debug my_mailer
```

Apprenez en plus

- *Compiler le Conteneur*
- *Travailler avec les définitions du conteneur*
- *Utiliser une « Factory » pour créer des services*
- *Gérer les dépendances communes avec des services parents*
- *Travailler avec des Services Taggés*
- *Comment définir des contrôleurs en tant que Services*
- *Comment travailler avec les champs d'applications (« scopes » en anglais)*
- *Comment travailler avec les Passes de Compilation dans les Bundles*
- *Configuration Avancée du Conteneur*

Chapter 17
Performance

Symfony2 est rapide, sans aucune configuration. Bien entendu, si vous voulez que ça soit plus rapide, il y a de nombreuses façons de rendre Symfony encore plus rapide. Dans ce chapitre, vous explorerez les méthodes les plus courantes et les plus puissantes pour rendre votre application Symfony encore plus rapide.

Utiliser un cache de byte code (ex APC)

L'une des meilleures choses (et des plus simples) que vous devriez faire pour augmenter la performance est d'utiliser un cache de byte code (« byte code cache »). L'idée d'un cache de byte code est de ne plus avoir à recompiler constamment le code source PHP. Il existe un certain nombre de *caches de byte code*, dont certains sont open source. Le cache de byte code le plus largement utilisé est probablement *APC*[1]

Il n'y a aucun effet négatif à utiliser un cache de byte code, et Symfony2 a été conçu pour être vraiment performant dans ce type d'environnement.

D'autres optimisations

Le cache de byte code surveille normalement tout changement de code source. Cela permet de recompiler automatiquement le byte code en cas de changement d'un fichier source. C'est très pratique, mais cela a évidemment un coût.

Pour cette raison, certains caches de byte code offrent l'option de désactiver la vérification, ce sera donc à l'administrateur système de vider le cache à chaque changement de fichier. Sinon, les mises à jour ne seront pas visibles.

Par exemple, pour désactiver cette vérification avec APC, ajoutez simplement `apc.stat=0` à votre fichier de configuration php.ini.

1. `http://php.net/manual/en/book.apc.php`

Utiliser un chargeur automatique qui met en cache (par exemple `ApcUniversalClassLoader`)

Par défaut, la distribution Symfony standard utilise `UniversalClassLoader` dans le fichier *autoloader.php*[2]. Ce chargeur automatique (autoloader) est facile à utiliser, car il va automatiquement trouver toute nouvelle classe que vous aurez placée dans les répertoires enregistrés.

Malheureusement, cela a un coût, et le chargeur itère à travers tous les espaces de noms (namespaces) pour trouver un fichier en particulier, en appelant `file_exists` jusqu'à ce qu'il trouve le fichier qu'il recherchait.

La solution la plus simple est de mettre en cache l'emplacement de chaque classe après qu'elles ont été trouvées la première fois. Symfony est fourni avec une classe - *`ApcClassLoader`*[3] - qui fait exactement cela. Pour l'utiliser, il vous suffit d'adapter votre contrôleur frontal. Si vous utilisez la distribution standard, le code devrait déjà être disponible dans ce fichier en commentaires:

Listing 17-1
```php
// app.php
// ...

$loader = require_once __DIR__.'/../app/bootstrap.php.cache';

// Use APC for autoloading to improve performance
// Change 'sf2' by the prefix you want in order to prevent key conflict with another
application
/*
$loader = new ApcClassLoader('sf2', $loader);
$loader->register(true);
*/

// ...
```

Lorsque vous utilisez le chargeur automatique APC, si vous avez de nouvelles classes, elles vont être trouvées automatiquement et tout fonctionnera comme avant (donc aucune raison de vider le cache). Par contre, si vous changez l'emplacement d'un espace de noms ou d'un préfixe, il va falloir que vous vidiez le cache APC. Sinon, le chargeur automatique verra toujours l'ancien emplacement des classes à l'intérieur de ce namespace.

Utiliser des fichiers d'amorçage

Afin d'assurer une flexibilité maximale et de favoriser la réutilisation du code, les applications Symfony2 profitent de nombreuses classes et composants externes. Mais charger toutes les classes depuis des fichiers séparés à chaque requête peut entraîner des coûts. Afin de réduire ces coûts, la distribution Symfony standard fournit un script qui génère ce qu'on appelle un *fichier d'amorçage*[4] (« bootstrap file »), qui consiste en une multitude de définitions de classes en un seul fichier. En incluant ce fichier (qui contient une copie de nombreux fichiers du core), Symfony n'a plus besoin d'inclure les fichiers sources individuels qui contiennent ces classes. Cela va réduire un peu les lectures/écritures sur le disque.

Si vous utilisez l'édition Symfony standard, alors vous utilisez probablement déjà le fichier d'amorçage. Pour vous en assurer, ouvrez votre contrôleur frontal (généralement `app.php`) et vérifiez que la ligne suivante est présente :

2. https://github.com/symfony/symfony-standard/blob/2.2/app/autoload.php
3. http://api.symfony.com/master/Symfony/Component/ClassLoader/ApcClassLoader.html
4. https://github.com/sensio/SensioDistributionBundle/blob/master/Composer/ScriptHandler.php

Listing 17-2
```php
require_once __DIR__.'/../app/bootstrap.php.cache';
```

Veuillez noter qu'il y a deux inconvénients à utiliser un fichier d'amorçage :

- le fichier nécessite d'être régénéré à chaque fois que les fichiers sources originaux changent (à savoir quand vous mettez à jour le code source de Symfony2 ou une bibliothèque tierce),
- lors du débogage, vous devrez placer des points d'arrêt (breakpoints) dans ce fichier d'amorçage.

Si vous utilisez l'édition Symfony2 standard, les fichiers d'amorçage sont automatiquement régénérés après avoir mis à jour les bibliothèques tierces (« vendors ») grâce à la commande `php bin/vendors install`.

Fichiers d'amorçage et caches de byte code

Même en utilisant un cache de byte code, la performance sera améliorée si vous utilisez un fichier d'amorçage, car il y aura moins de fichiers dont il faut surveiller les changements. Bien sûr, si vous désactivez la surveillance des changements de fichiers (par exemple `apc.stat=0` in APC), il n'y a plus de raison d'utiliser un fichier d'amorçage.

Chapter 18
Composants Internes

Visiblement vous voulez comprendre comment Symfony2 fonctionne et comment l'étendre. Cela me rend très heureux ! Cette section explique en profondeur les composants internes de Symfony2.

Vous avez besoin de lire cette section uniquement si vous souhaitez comprendre comment Symfony2 fonctionne en arrière-plan, ou si vous voulez étendre Symfony2.

Vue Globale

Le code de Symfony2 se compose de plusieurs couches indépendantes. Chacune d'entre elles est construite par-dessus celles qui la précèdent.

L'autoloading (« chargement automatique ») n'est pas géré par le framework directement ; ceci est effectué indépendamment à l'aide de la classe *`UniversalClassLoader`*[1] et du fichier `src/autoload.php`. Lisez le *chapitre dédié* pour plus d'informations.

Le Composant `HttpFoundation`

Le composant de plus bas niveau est *`HttpFoundation`*[2]. HttpFoundation fournit les objets principaux nécessaires pour traiter avec HTTP. C'est une abstraction orientée objet de quelques fonctions et variables PHP natives :

- La classe *`Request`*[3] abstrait les principales variables globales PHP que sont `$_GET`, `$_POST`, `$_COOKIE`, `$_FILES`, et `$_SERVER` ;
- Le classe *`Response`*[4] abstrait quelques fonctions PHP comme `header()`, `setcookie()`, et `echo` ;

1. http://api.symfony.com/master/Symfony/Component/ClassLoader/UniversalClassLoader.html
2. http://api.symfony.com/master/Symfony/Component/HttpFoundation.html
3. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Request.html
4. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html

- La classe *Session*[5] et l'interface *SessionStorageInterface*[6] font abstraction des fonctions de gestion des sessions `session_*()`.

Le Composant `HttpKernel`

Par-dessus HttpFoundation se trouve le composant *HttpKernel*[7]. HttpKernel gère la partie dynamique de HTTP ; c'est une fine surcouche au-dessus des classes Request et Response pour standardiser la façon dont les requêtes sont gérées. Il fournit aussi des points d'extension et des outils qui en font un point de démarrage idéal pour créer un framework Web sans trop d'efforts.

Optionnellement, il ajoute de la configurabilité et de l'extensibilité, grâce au composant Dependency Injection et à un puissant système de plugin (bundles).

Apprenez en plus en lisant les chapitres Dependency Injection et Bundles.

Le Bundle `FrameworkBundle`

Le bundle *FrameworkBundle*[8] est le bundle qui lie les principaux composants et bibliothèques ensembles afin de fournir un framework MVC léger et rapide. Il est fourni avec une configuration par défaut ainsi qu'avec des conventions afin d'en faciliter l'apprentissage.

Le Kernel

La classe *HttpKernel*[9] est la classe centrale de Symfony2 et est responsable de la gestion des requêtes clientes. Son but principal est de « convertir » un objet *Request*[10] en un objet *Response*[11].

Chaque Kernel Symfony2 implémente *HttpKernelInterface*[12]

Listing 18-1
```php
function handle(Request $request, $type = self::MASTER_REQUEST, $catch = true)
```

Les Contrôleurs

Pour convertir une Requête en une Réponse, le Kernel repose sur un « Contrôleur ». Un Contrôleur peut être n'importe quel « callable » (code qui peut être appelé) PHP.

Le Kernel délègue la sélection de quel Contrôleur devrait être exécuté à une implémentation de *ControllerResolverInterface*[13]

Listing 18-2
```php
public function getController(Request $request);

public function getArguments(Request $request, $controller);
```

La méthode *getController()*[14] retourne le Contrôleur (un « callable » PHP) associé à la Requête donnée. L'implémentation par défaut (*ControllerResolver*[15]) recherche un attribut de la requête

5. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Session.html
6. http://api.symfony.com/master/Symfony/Component/HttpFoundation/SessionStorage/SessionStorageInterface.html
7. http://api.symfony.com/master/Symfony/Component/HttpKernel.html
8. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle.html
9. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpKernel.html
10. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Request.html
11. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Response.html
12. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpKernelInterface.html
13. http://api.symfony.com/master/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html
14. http://api.symfony.com/master/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getController()

`_controller` qui représente le nom du contrôleur (une chaîne de caractères « classe::méthode », comme `Bundle\BlogBundle\PostController:indexAction`).

L'implémentation par défaut utilise le *`RouterListener`*[16] pour définir l'attribut de la Requête `_controller` (voir *L'Évènement kernel.request*).

La méthode *`getArguments()`*[17] retourne un tableau d'arguments à passer au Contrôleur. L'implémentation par défaut résout automatiquement les arguments de la méthode, basée sur les attributs de la Requête.

Faire correspondre les arguments de la méthode du Contrôleur aux attributs de la Requête

Pour chaque argument d'une méthode, Symfony2 essaie d'obtenir la valeur d'un attribut d'une Requête avec le même nom. S'il n'est pas défini, la valeur par défaut de l'argument est utilisée si elle est définie

Listing 18-3
```php
// Symfony2 va rechercher un attribut « id » (obligatoire)
// et un nommé « admin » (optionnel)
public function showAction($id, $admin = true)
{
    // ...
}
```

Gestion des Requêtes

La méthode *`handle()`*[18] prend une `Requête` et retourne *toujours* une `Réponse`. Pour convertir la `Requête`, `handle()` repose sur le « Resolver » et sur une chaîne ordonnée de notifications d'évènements (voir la prochaine section pour plus d'informations à propos de chaque évènement) :

1. Avant de faire quoi que ce soit d'autre, l'évènement `kernel.request` est notifié -- si l'un des listeners (« écouteurs » en français) retourne une `Réponse`, il saute directement à l'étape 8 ;
2. Le « Resolver » est appelé pour déterminer le Contrôleur à exécuter ;
3. Les listeners de l'évènement `kernel.controller` peuvent maintenant manipuler le « callable » Contrôleur de la manière dont ils souhaitent (le changer, créer un « wrapper » au-dessus de lui, ...) ;
4. Le Kernel vérifie que le Contrôleur est un « callable » PHP valide ;
5. Le « Resolver » est appelé pour déterminer les arguments à passer au Contrôleur ;
6. Le Kernel appelle le Contrôleur ;
7. Si le Contrôleur ne retourne pas une `Réponse`, les listeners de l'évènement `kernel.view` peuvent convertir la valeur retournée par le Contrôleur en une `Réponse` ;
8. Les listeners de l'évènement `kernel.response` peuvent manipuler la `Réponse` (contenu et en-têtes) ;
9. La Réponse est retournée.

Si une Exception est capturée pendant le traitement de la Requête, l'évènement `kernel.exception` est notifié et les listeners ont alors une chance de convertir l'Exception en une Réponse. Si cela fonctionne, l'évènement `kernel.response` sera notifié ; sinon, l'Exception sera rejetée.

15. http://api.symfony.com/master/Symfony/Component/HttpKernel/Controller/ControllerResolver.html
16. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle/EventListener/RouterListener.html
17. http://api.symfony.com/master/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getArguments()
18. http://api.symfony.com/master/Symfony/Component/HttpKernel/HttpKernel.html#handle()

Si vous ne voulez pas que les Exceptions soient capturées (pour des requêtes imbriquées par exemple), désactivez l'évènement `kernel.exception` en passant `false` en tant que troisième argument de la méthode `handle()`.

Requêtes Internes

A tout moment, durant la gestion de la requête (la « master »), une sous-requête peut être gérée. Vous pouvez passer le type de requête à la méthode `handle()` (son second argument) :

- `HttpKernelInterface::MASTER_REQUEST`;
- `HttpKernelInterface::SUB_REQUEST`.

Le type est passé à tous les évènements et les listeners peuvent ainsi agir en conséquence (le traitement doit seulement intervenir sur la requête « master »).

Les Évènements

Chaque évènement capturé par le Kernel est une sous-classe de *`KernelEvent`*[19]. Cela signifie que chaque évènement a accès aux mêmes informations de base :

- *`getRequestType`()*[20] - retourne le *type* de la requête (`HttpKernelInterface::MASTER_REQUEST` ou `HttpKernelInterface::SUB_REQUEST`) ;
- *`getKernel()`*[21] - retourne le Kernel gérant la requête ;
- *`getRequest()`*[22] - retourne la `Requête` courante qui est en train d'être gérée.

`getRequestType()`

La méthode `getRequestType()` permet aux listeners de connaître le type de la requête. Par exemple, si un listener doit seulement être activé pour les requêtes « master », ajoutez le code suivant au début de votre méthode listener

Listing 18-4
```php
use Symfony\Component\HttpKernel\HttpKernelInterface;

if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
    // retourne immédiatement
    return;
}
```

Si vous n'êtes pas encore familier avec le « Dispatcher d'Évènements » de Symfony2, lisez d'abord la section *Documentation du composant Event Dispatcher*.

L'Évènement `kernel.request`

La Classe Évènement : *`GetResponseEvent`*[23]

Le but de cet évènement est soit de retourner un objet `Response` immédiatement, soit de définir des variables afin qu'un Contrôleur puisse être appelé après l'évènement. Tout listener peut retourner un objet `Response` via la méthode `setResponse()` sur l'évènement. Dans ce cas, tous les autres listeners ne seront pas appelés.

19. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/KernelEvent.html
20. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/KernelEvent.html#getRequestType`()
21. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/KernelEvent.html#getKernel()
22. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/KernelEvent.html#getRequest()
23. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/GetResponseEvent.html

Cet évènement est utilisé par le `FrameworkBundle` afin de remplir l'attribut de la `Requête` `_controller`, via *`RouterListener`*[24]. RequestListener utilise un objet *`RouterInterface`*[25] pour faire correspondre la `Requête` et déterminer le nom du Contrôleur (stocké dans l'attribut de la `Requête` `_controller`).

L'évènement `kernel.controller`

La Classe Évènement: *`FilterControllerEvent`*[26]

Cet évènement n'est pas utilisé par le `FrameworkBundle`, mais peut être un point d'entrée utilisé pour modifier le contrôleur qui devrait être exécuté:

Listing 18-5
```php
use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

public function onKernelController(FilterControllerEvent $event)
{
    $controller = $event->getController();
    // ...

    // le contrôleur peut être remplacé par n'importe quel « callable » PHP
    $event->setController($controller);
}
```

L'évènement `kernel.view`

La Classe Évènement: *`GetResponseForControllerResultEvent`*[27]

Cet évènement n'est pas utilisé par le `FrameworkBundle`, mais il peut être utilisé pour implémenter un sous-système de vues. Cet évènement est appelé *seulement* si le Contrôleur *ne* retourne *pas* un objet `Response`. Le but de cet évènement est de permettre à d'autres valeurs retournées d'être converties en une `Réponse`.

La valeur retournée par le Contrôleur est accessible via la méthode `getControllerResult`

Listing 18-6
```php
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelView(GetResponseForControllerResultEvent $event)
{
    $val = $event->getReturnValue();
    $response = new Response();
    // personnalisez d'une manière ou d'une autre la Réponse
    // en vous basant sur la valeur retournée

    $event->setResponse($response);
}
```

L'évènement `kernel.response`

La Classe Évènement: *`FilterResponseEvent`*[28]

L'objectif de cet évènement est de permettre à d'autres systèmes de modifier ou de remplacer l'objet `Response` après sa création :

24. http://api.symfony.com/master/Symfony/Bundle/FrameworkBundle/EventListener/RouterListener.html
25. http://api.symfony.com/master/Symfony/Component/Routing/RouterInterface.html
26. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html
27. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/GetResponseForControllerResultEvent.html
28. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html

Listing 18-7
```php
public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    // .. modifiez l'objet Response
}
```

Le `FrameworkBundle` enregistre plusieurs listeners :

- *`ProfilerListener`*[29]: collecte les données pour la requête courante ;
- *`WebDebugToolbarListener`*[30]: injecte la Barre d'Outils de Débuggage Web (« Web Debug Toolbar ») ;
- *`ResponseListener`*[31]: définit la valeur du `Content-Type` de la Réponse basée sur le format de la requête ;
- *`EsiListener`*[32]: ajoute un en-tête HTTP `Surrogate-Control` lorsque la Réponse a besoin d'être analysée pour trouver des balises ESI.

L'évènement `kernel.exception`

La Classe Évènement: *`GetResponseForExceptionEvent`*[33]

Le `FrameworkBundle` enregistre un *`ExceptionListener`*[34] qui transmet la `Requête` à un Contrôleur donné (la valeur du paramètre `exception_listener.controller` -- doit être exprimé suivant la notation `class::method`).

Un listener sur cet évènement peut créer et définir un objet `Response`, créer et définir un nouvel objet `Exception`, ou ne rien faire :

Listing 18-8
```php
use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelException(GetResponseForExceptionEvent $event)
{
    $exception = $event->getException();
    $response = new Response();
    // définissez l'objet Response basé sur l'exception capturée
    $event->setResponse($response);

    // vous pouvez alternativement définir une nouvelle Exception
    // $exception = new \Exception('Some special exception');
    // $event->setException($exception);
}
```

Comme Symfony vérifie que le code de statut de la Réponse est le plus approprié selon l'exception, définir un code de statut sur la réponse ne fonctionnera pas. Si vous voulez réécrire le code de statut (ce que vous ne devriez pas faire sans une excellente raison), définissez l'entête `X-Status-Code`:

Listing 18-9
```php
return new Response('Error', 404 /* ignoré */, array('X-Status-Code' => 200));
```

29. http://api.symfony.com/master/Symfony/Component/HttpKernel/EventListener/ProfilerListener.html
30. http://api.symfony.com/master/Symfony/Bundle/WebProfilerBundle/EventListener/WebDebugToolbarListener.html
31. http://api.symfony.com/master/Symfony/Component/HttpKernel/EventListener/ResponseListener.html
32. http://api.symfony.com/master/Symfony/Component/HttpKernel/EventListener/EsiListener.html
33. http://api.symfony.com/master/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
34. http://api.symfony.com/master/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html

Le Dispatcher d'Évènements

Le dispatcher d'évènements est un composant autonome qui est responsable d'une bonne partie de la logique sous-jacente et du flux d'une requête Symfony. Pour plus d'informations, lisez-la *documentation du composant Event Dispatcher*.

Profiler

Lorsqu'il est activé, le profiler de Symfony2 collecte des informations utiles concernant chaque requête envoyée à votre application et les stocke pour une analyse future. Utilisez le profiler dans l'environnement de développement afin de vous aider à débugger votre code et à améliorer les performances de votre application; utilisez-le dans l'environnement de production pour explorer des problèmes après coup.

Vous avez rarement besoin d'interagir avec le profiler directement puisque Symfony2 vous fournit des outils de visualisation tels la Barre d'Outils de Débuggage Web (« Web Debug Toolbar ») et le Profiler Web (« Web Profiler »). Si vous utilisez l'Edition Standard de Symfony2, le profiler, la barre d'outils de débuggage web, et le profiler web sont tous déjà configurés avec des paramètres prédéfinis.

Le profiler collecte des informations pour toutes les requêtes (simples requêtes, redirections, exceptions, requêtes Ajax, requêtes ESI; et pour toutes les méthodes HTTP et tous les formats). Cela signifie que pour une même URL, vous pouvez avoir plusieurs données de profiling associées (une par paire de requête/réponse externe).

Visualiser les Données de Profiling

Utiliser la Barre d'Outils de Débuggage Web

Dans l'environnement de développement, la barre d'outils de débuggage web est disponible en bas de toutes les pages. Elle affiche un bon résumé des données de profiling qui vous donne accès instantanément à plein d'informations utiles quand quelque chose ne fonctionne pas comme prévu.

Si le résumé fourni par la Barre d'Outils de Débuggage Web n'est pas suffisant, cliquez sur le lien du jeton (une chaîne de caractères composée de 13 caractères aléatoires) pour pouvoir accéder au Profiler Web.

Si le jeton n'est pas cliquable, cela signifie que les routes du profiler ne sont pas enregistrées (voir ci-dessous pour les informations concernant la configuration).

Analyser les données de Profiling avec le Profiler Web

Le Profiler Web est un outil de visualisation pour profiler des données que vous pouvez utiliser en développement pour débugger votre code et améliorer les performances ; mais il peut aussi être utilisé pour explorer des problèmes qui surviennent en production. Il expose toutes les informations collectées par le profiler via une interface web.

Accéder aux informations de Profiling

Vous n'avez pas besoin d'utiliser l'outil de visualisation par défaut pour accéder aux informations de profiling. Mais comment pouvez-vous obtenir les informations de profiling pour une requête spécifique après coup ? Lorsque le profiler stocke les données concernant une Requête, il lui associe aussi un jeton ; ce jeton est disponible dans l'en-tête HTTP `X-Debug-Token` de la Réponse

Listing 18-10

```php
$profile = $container->get('profiler')->loadProfileFromResponse($response);

$profile = $container->get('profiler')->loadProfile($token);
```

Lorsque le profiler est activé, mais sans la barre d'outils de débuggage web, ou lorsque vous voulez récupérer le jeton pour une requête Ajax, utilisez un outil comme Firebug pour obtenir la valeur de l'en-tête HTTP `X-Debug-Token`.

Utilisez la méthode *`find()`*[35] pour accéder aux jetons basés sur quelques critères :

// récupère les 10 derniers jetons $tokens = $container->get('profiler')->find('', '', 10);

// récupère les 10 derniers jetons pour toutes les URL contenant /admin/ $tokens = $container->get('profiler')->find('', '/admin/', 10);

// récupère les 10 derniers jetons pour les requêtes locales $tokens = $container->get('profiler')->find('127.0.0.1', '', 10);

Si vous souhaitez manipuler les données de profiling sur une machine différente que celle où les informations ont été générées, utilisez les méthodes *`export()`*[36] et *`import()`*[37]:

Listing 18-11
```php
// sur la machine de production
$profile = $container->get('profiler')->loadProfile($token);
$data = $profiler->export($profile);

// sur la machine de développement
$profiler->import($data);
```

Configuration

La configuration par défaut de Symfony2 vient avec des paramètres prédéfinis pour le profiler, la barre d'outils de débuggage web, et le profiler web. Voici par exemple la configuration pour l'environnement de développement :

Listing 18-12
```yaml
# charge le profiler
framework:
    profiler: { only_exceptions: false }

# active le profiler web
web_profiler:
    toolbar: true
    intercept_redirects: true
```

Quand l'option `only-exceptions` est définie comme `true`, le profiler collecte uniquement des données lorsqu'une exception est capturée par l'application.

Quand l'option `intercept-redirects` est définie à `true`, le web profiler intercepte les redirections et vous donne l'opportunité d'inspecter les données collectées avant de suivre la redirection.

Si vous activez le profiler web, vous avez aussi besoin de monter les routes du profiler :

Listing 18-13

35. http://api.symfony.com/master/Symfony/Component/HttpKernel/Profiler/Profiler.html#find()
36. http://api.symfony.com/master/Symfony/Component/HttpKernel/Profiler/Profiler.html#export()
37. http://api.symfony.com/master/Symfony/Component/HttpKernel/Profiler/Profiler.html#import()

```
_profiler:
    resource: @WebProfilerBundle/Resources/config/routing/profiler.xml
    prefix:   /_profiler
```

Comme le profiler rajoute du traitement supplémentaire, vous pourriez vouloir l'activer uniquement selon certaines circonstances dans l'environnement de production. Le paramètre `only-exceptions` limite le profiling aux pages 500, mais qu'en est-il si vous voulez avoir les informations lorsque l'IP du client provient d'une adresse spécifique, ou pour une portion limitée du site web ? Vous pouvez utiliser la correspondance de requête :

Listing 18-14
```yaml
# active le profiler uniquement pour les requêtes venant du réseau 192.168.0.0
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24 }

# active le profiler uniquement pour les URLs /admin
framework:
    profiler:
        matcher: { path: "^/admin/" }

# associe des règles
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24, path: "^/admin/" }

# utilise une instance de correspondance personnalisée définie dans le
# service "custom_matcher"
framework:
    profiler:
        matcher: { service: custom_matcher }
```

En savoir plus grâce au Cookbook

- *Comment utiliser le Profiler dans un test fonctionnel*
- *Comment créer un Collecteur de Données personnalisé*
- *Comment étendre une Classe sans utiliser l'Héritage*
- *Comment personnaliser le Comportement d'une Méthode sans utiliser l'Héritage*

Chapter 19
L'API stable de Symfony2

L'API stable de Symfony2 est un sous-ensemble de toutes les méthodes publiques de Symfony2 (composants et bundles "du coeur") qui partagent les propriétés suivantes:

- le namespace et le nom de la classe ne changeront pas,
- le nom de la méthode ne changera pas,
- la signature de la méthode (arguments et valeur de retour) ne changera pas,
- l'objectif de la méthode ne changera pas

L'implémentation elle-même peut changer. La seule raison valable d'un changement de l'API stable de Symfony2 serait de corriger un problème de sécurité.

L'API stable est basée sur un principe de liste blanche ("whitelist"), taggée par *@api*. En conséquence, tout ce qui n'est pas explicitement tagué ne fait pas partie de l'API stable.

Chaque bundle tiers devrait aussi publier sa propre API stable.

Tout comme Symfony 2.0, les composants suivants ont leur propre API publique :

- BrowserKit
- ClassLoader
- Console
- CssSelector
- DependencyInjection
- DomCrawler
- EventDispatcher
- Finder
- HttpFoundation
- HttpKernel
- Locale
- Process
- Routing
- Templating
- Translation

- Validator
- Yaml