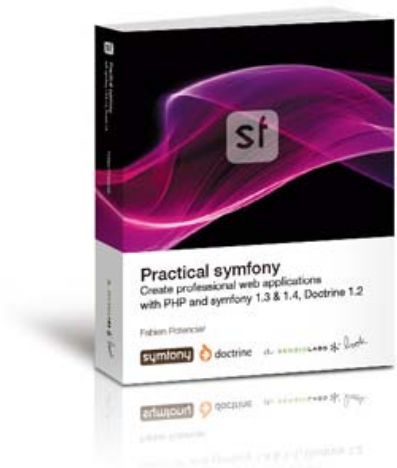
Practical symfony

"Practical symfony" is part of the official symfony documentation.



Practical symfony
Jour 1 : Démarrage du projet

Introduction

Le framework symfony est un projet Open-Source depuis plus de quatre ans et est devenu l'un des plus populaires framework PHP grâce à ses excellentes fonctionnalités et sa très bonne documentation.

Ce livre décrit la création d'une application web avec le framework symfony, étape par étape, des spécifications à la mise en œuvre. Il s'adresse aux débutants qui veulent apprendre symfony, comprendre comment il fonctionne, et aussi se renseigner sur les meilleures pratiques de développement Web.

L'application à réaliser aurait pu être un autre moteur de blog. Mais nous souhaitons utiliser symfony sur un projet utile. L'objectif est de démontrer que symfony peut être utilisé pour développer des applications professionnelles avec style et peu d'effort.

Nous tiendrons le contenu du projet secret pour un autre jour car nous avons déjà beaucoup à faire aujourd'hui. Cependant, donnons-lui un nom : **Jobeet**.

Chaque chapitre de ce livre est fait pour durer entre une et deux heures, et sera l'occasion d'apprendre symfony en codant un vrai site web, du début à la fin. Chaque jour, de nouvelles fonctionnalités seront ajoutées à l'application, et nous allons profiter de ce développement pour vous présenter de nouvelles fonctionnalités de symfony, ainsi que les bonnes pratiques dans le développement web de symfony.

Ce livre est différent

Rappelez-vous les premiers jours de PHP4. Ah, la Belle Epoque ! PHP est l'un des premiers langages dédiés au web et l'un des plus facile à apprendre.

Mais comme les technologies du web évoluent à un rythme très rapide, les développeurs Web doivent se tenir au courant sur les dernières meilleures pratiques et les outils. La meilleure façon d'apprendre est bien sûr par la lecture des blogs, des tutoriels et des livres. Nous avons lu beaucoup d'entre eux, qu'ils soient écrits pour PHP, Python, Java, Ruby ou Perl, et beaucoup d'entre eux nous laisse sur notre faim lorsque l'auteur commence à donner des morceaux de codes à titre d'exemples.

Vous êtes sans doute habitué à lire des avertissements du genre:

«Pour une application réelle, n'oubliez pas d'ajouter la validation et la manipulation d'erreur appropriée.»

Ou

«La sécurité est laissée en exercice au lecteur.»

Ou

«Vous aurez bien sûr besoin d'écrire des tests.»

Pardon ? Ces choses sont sérieuses. Ils sont peut-être la partie la plus importante de tout morceau de code. Et en tant que lecteur, on vous laisse seuls. Sans que ces préoccupations soient prises en compte, les exemples sont beaucoup moins utiles. Vous ne pouvez pas les utiliser comme un bon point de départ. C'est très grave ! Pourquoi ? Parce que la sécurité, la validation, le traitement des erreurs et les tests, pour n'en nommer que quelques-uns, sont importants pour avoir un bon code.

Dans ce livre, vous ne verrez jamais des déclarations car nous allons écrire des tests, la gestion d'erreur, le code de validation, et soyez sûr que nous développons une application sécurisée. C'est parce que symfony est sur??? le code, mais également sur les meilleures pratiques et comment développer des applications professionnelles pour l'entreprise. Nous serons en mesure de se permettre ce luxe, car symfony fournit tous les outils nécessaires pour coder ces aspects

aisément sans écrire trop de code.

La validation, la manipulation d'erreur, la sécurité, et les tests sont des citoyens de première classe dans symfony, ainsi cela ne nous prendra pas trop de temps à expliquer. Ce n'est là qu'une des nombreuses raisons d'utiliser un framework pour la «vie réelle» des projets.

Tout le code que vous lirez dans ce livre est un code que vous pourriez utiliser pour un projet réel. Nous vous encourageons à copier/coller des bouts de code ou de voler des morceaux entiers.

Qu'allons nous faire aujourd'hui ?

Nous n'allons pas écrire du code PHP aujourd'hui. Mais même sans écrire une seule ligne de code, vous commencerez à comprendre les avantages de l'utilisation d'un framework comme symfony, juste en démarrant un nouveau projet.

L'objectif de ce chapitre est d'installer l'environnement de développement et afficher une page de l'application dans un navigateur web. Cela comprend l'installation de symfony, la création d'une application et la configuration du serveur web.

Comme ce livre met principalement l'accent sur le framework symfony, nous supposerons que vous avez déjà une solide connaissance en PHP 5 et en programmation orientée objet.

Prérequis

Avant d'installer symfony, vous devez vérifier que votre ordinateur a tout installé et configuré correctement. Prenez le temps de lire consciencieusement ce chapitre et de suivre toutes les étapes nécessaires pour vérifier votre configuration, car elle peut sauver votre journée plus loin sur le parcours.

Logiciel tiers

Tout d'abord, vous devez vérifier que votre ordinateur dispose d'un environnement de travail convivial pour le développement web. Au minimum, vous avez besoin d'un serveur web (Apache, par exemple), un moteur de base de données (MySQL, PostgreSQL, SQLite, ou tout autre moteur de base de PDO-compatible), et PHP 5.2.4 ou ultérieur.

Interface de ligne de commande

Le framework symfony est livré avec un outil de ligne de commande qui permet d'automatiser beaucoup de travail pour vous. Si vous êtes un utilisateur d'un OS de type Unix, vous vous sentirez comme chez vous. Si vous utilisez un système Windows, il fonctionne également très bien, mais il vous suffira de taper quelques commandes à l'invite de `cmd`.

Les commandes shell Unix peuvent être pratiques dans un environnement Windows. Si vous souhaitez utiliser des outils comme `tar`, `gzip` ou `grep` sous Windows, vous pouvez installer Cygwin. Les aventureux peuvent aussi essayer les Windows Services for Unix de Microsoft.

Configuration de PHP

Comme les configurations PHP peuvent beaucoup varier d'un OS à l'autre, ou même entre les différentes distributions Linux, vous devez vérifier que votre configuration de PHP satisfait aux exigences minimales de symfony.

Premièrement, assurez-vous que PHP 5.2.4 au minimum est installé en utilisant la fonction intégrée `phpinfo()` ou en exécutant `php -v` en ligne de commande. Soyez conscient que sur certaines configurations, vous pouvez avoir deux versions différentes de PHP d'installées : l'une pour la ligne de commande, et une autre pour le web.

Ensuite, téléchargez le script de vérification de la configuration de symfony à l'adresse suivante :

```
http://sf-to.org/1.4/check.php
```

Enregistrez le script quelque part sous votre répertoire racine web.

Lancez le script de vérification de configuration en ligne de commande :

```
$ php check_configuration.php
```

S'il y a un problème avec votre configuration PHP, la sortie de la commande va vous donner des conseils sur ce qu'il faut corriger et comment le réparer.

Vous devez également exécuter la vérification à partir d'un navigateur et corriger les problèmes

qu'elle pourrait découvrir. Car PHP peut avoir un fichier de configuration `php.ini` distinct pour ces deux environnements, avec des réglages différents.

N'oubliez pas de supprimer le fichier de votre répertoire racine web par la suite.

Installation de symfony

L'initialisation du répertoire du projet

Avant d'installer symfony, vous devez d'abord créer un répertoire qui sera l'hôte de tous les fichiers liés à Jobeet :

```
$ mkdir -p /home/sfprojects/jobeet
$ cd /home/sfprojects/jobeet
```

Ou sous Windows:

```
c:\> mkdir c:\development\sfprojects\jobeet
c:\> cd c:\development\sfprojects\jobeet
```

Les utilisateurs de Windows sont invités à exécuter symfony et mettre en place leur nouveau projet dans un chemin qui ne contient pas d'espaces. Évitez d'utiliser le répertoire `Documents and Settings`, y compris n'importe où sous `Mes documents`.

Si vous créez le répertoire du projet symfony sous le répertoire racine web, vous n'aurez pas besoin de configurer votre serveur web. Bien sûr, pour les environnements de production, nous vous conseillons fortement de configurer votre serveur web, comme expliqué dans la section de configuration du serveur web.

Choix de la version de symfony

Maintenant, vous devez installer symfony. Comme le framework symfony dispose de plusieurs versions stables, vous devez choisir celle que vous souhaitez installer en lisant la page d'installation sur le site de symfony.

Ce livre suppose que vous voulez installer symfony 1.3 ou symfony 1.4.

Choix du lieu d'installation de symfony

Vous pouvez installer symfony globalement sur votre machine, ou l'intégrer à chacun de vos projets. Cette dernière est celle qui est recommandée car les projets seront alors totalement indépendants les uns des autres. La mise à jour de votre symfony installé localement ne cassera pas tous vos projets de manière inattendue. Cela signifie que vous pourrez avoir des projets sur les différentes versions de symfony, et les mettre à jour comme bon vous semble.

Comme bonne pratique, beaucoup de gens installent les fichiers du framework symfony dans le répertoire `lib/vendor` du projet. Donc, premièrement, créez ce répertoire :

```
$ mkdir -p lib/vendor
```

Installation de symfony

Installation depuis une archive

Le moyen le plus simple d'installer symfony est de télécharger l'archive pour la version voulue sur le site de symfony. Allez à la page d'installation pour la version que vous venez de choisir, symfony 1.4 par exemple.

Sous la section "**Source Download**", vous trouverez l'archive au format `.tgz` ou `.zip`. Téléchargez l'archive, mettez-la sous le répertoire `lib/vendor/` créé précédemment, décompressez-la, et renommez le répertoire en `symfony` :

```
$ cd lib/vendor
$ tar zxpf symfony-1.4.0.tgz
$ mv symfony-1.4.0 symfony
$ rm symfony-1.4.0.tgz
```

Sous Windows, décompressez le fichier zip qui peut être réalisé en utilisant l'explorateur de Windows. Après avoir renommé le répertoire en `symfony`, il devrait y avoir une structure de répertoire similaire à `c:\dev\sfprojects\jobeet\lib\vendor\symfony`.

Installation depuis Subversion (recommandée)

Si vous utilisez Subversion, il est même préférable d'utiliser la propriété `svn:externals` pour incorporer symfony dans votre projet dans le répertoire `lib/vendor/` directory :

```
$ svn pe svn:externals lib/vendor/
```

L'importation de votre projet dans un nouveau dépôt de Subversion est expliqué à la fin de ce chapitre.

Si tout va bien, cette commande va lancer votre éditeur favori pour vous donner la possibilité de configurer les sources extérieures de Subversion.

Sous Windows, vous pouvez utiliser des outils comme TortoiseSVN(http://tortoisesvn.net/) pour tout faire sans avoir besoin d'utiliser la console.

Si vous êtes conservateur, attachez votre projet à une version spécifique (un tag de subversion) :

```
symfony http://svn.symfony-project.com/tags/RELEASE_1_4_0
```

Chaque fois qu'une nouvelle version sortira (comme annoncé sur le blog de symfony), vous devrez modifier l'URL vers la nouvelle version.

Si vous souhaitez la version la plus avancée mais habituellement la plus risquée, utilisez la branche 1.4 :

```
symfony http://svn.symfony-project.com/branches/1.4/
```

L'utilisation de la branche permet à votre projet de bénéficier de corrections de bogues automatiquement chaque fois que vous exécutez un `svn update`.

Vérification de l'installation

Maintenant que symfony est installé, vérifiez que tout fonctionne en utilisant la ligne de commande de symfony pour afficher la version de symfony (notez la majuscule V) :

```
$ cd ../..
$ php lib/vendor/symfony/data/bin/symfony -V
```

Sur Windows:

```
c:\> cd ..\..
c:\> php lib\vendor\symfony\data\bin\symfony -V
```

Si vous êtes curieux de savoir ce que cet outil en ligne de commande peut faire pour vous, tapez `symfony` pour lister les options et les tâches disponibles :

```
$ php lib/vendor/symfony/data/bin/symfony
```

Sur Windows:

```
c:\> php lib\vendor\symfony\data\bin\symfony
```

La ligne de commande symfony est la meilleure amie du développeur. Elle fournit de nombreux utilitaires, qui permettent d'améliorer votre productivité au quotidien sur des activités comme le nettoyage du cache, la génération de code et bien plus encore.

Installation du projet

Dans symfony, les **applications** partageant le même modèle de données sont regroupées dans des **projets**. Pour la plupart des projets, vous avez deux applications différentes : un frontend et un backend.

Création du projet

Depuis le répertoire `sfprojects/jobeet`, exécutez la tâche symfony `generate:project` pour créer effectivement le projet symfony :

```
$ php lib/vendor/symfony/data/bin/symfony generate:project jobeet
```

Sur Windows:

```
c:\> php lib\vendor\symfony\data\bin\symfony generate:project jobeet
```

La tâche `generate:project` génère la structure par défaut des répertoires et les fichiers nécessaires pour un projet symfony :

Répertoire	Description
`apps/`	Accueille toutes les applications du projet
`cache/`	Les fichiers mis en cache par le framework
`config/`	Les fichiers de configuration du projet
`lib/`	Les bibliothèques et les classes du projet
`log/`	Les fichiers log du framework
`plugins/`	Les plugins installés
`test/`	Les fichiers de test unitaire et fonctionnel
`web/`	Le répertoire racine Web (voir ci-dessous)

Pourquoi symfony génère beaucoup de fichiers ? Un des principaux avantages d'un framework full-stack est de normaliser vos développements. Grâce à la structure par défaut des fichiers et des répertoires de symfony, tout développeur ayant une certaine connaissance de symfony peut prendre en charge la maintenance d'un projet symfony. En quelques minutes, il sera capable de plonger dans le code, de corriger des bugs, et d'ajouter de nouvelles fonctionnalités.

La tâche `generate:project` a également créé un raccourci `symfony` dans le répertoire racine du projet pour diminuer le nombre de caractères que vous allez écrire lors de l'exécution d'une tâche.

Ainsi, à partir de maintenant, au lieu d'utiliser le chemin complet pour le programme symfony, vous pouvez utiliser le raccourci `symfony`.

Création de l'application

Maintenant, créez l'application frontend en exécutant la tâche `generate:app` :

```
$ php symfony generate:app frontend
```

Parce que le raccourci symfony est exécutable, les utilisateurs Unix peuvent remplacer toutes les occurrences de '`php symfony`' par '`./symfony`' à partir de maintenant.

Sur Windows vous pouvez copier le fichier '`symfony.bat`' vers votre projet et utilisez '`symfony`' à la place de '`php symfony`' :

```
c:\> copy lib\vendor\symfony\data\bin\symfony.bat .
```

Basé sur le nom de l'application donné en *argument*, la tâche `generate:app` crée par défaut la structure du répertoire nécessaire à l'application sous le répertoire `apps/frontend/` :

Répertoire	Description
`config/`	Les fichiers de configuration de l'application
`lib/`	Les bibliothèques et les classes de l'application
`modules/`	Le code de l'application (MVC)
`templates/`	Les fichiers template globaux

Securité

Par défaut, la tâche `generate:app` a sécurisé notre application sur les deux vulnérabilités les plus répandues que l'on trouve sur le web. C'est vrai, symfony prend automatiquement des mesures de sécurité à notre place.

Pour prévenir des attaques XSS, l'output escaping a été activé; et pour prévenir des attaques CSRF, un secret CSRF a été créé aléatoirement.

Bien sûr, vous pouvez modifier ces paramètres grâce aux *options* suivantes :

- `--escaping-strategy`: Active ou désactive l'output escaping
- `--csrf-secret`: active les jetons de session dans les formulaires

Si vous ne savez rien sur XSS ou CSRF, prenez le temps d'en apprendre d'avantage sur ces failles de sécurité.

Droits sur les répertoires structurés

Avant d'essayer d'accéder à votre projet nouvellement créé, vous devez configurer l'écriture sur les répertoires `cache/` et `log/` à des niveaux appropriés, afin que votre serveur web puisse écrire dedans :

```
$ chmod 777 cache/ log/
```

Conseils pour les personnes utilisant un outil de SCM

symfony écrit seulement dans deux répertoires du projet symfony : `cache/` et `log/`. Le contenu de ces répertoires peut-être ignoré par votre SCM (En utilisant la propriété `svn:ignore`, si vous utilisez Subversion par exemple).

Configuration du serveur Web : la pire méthode

Si vous avez créé le répertoire du projet quelque part dans le répertoire racine web de votre serveur web, vous pouvez déjà accéder au projet dans un navigateur Web.

Bien sûr, comme il n'y a pas de configuration, il est très rapide à mettre en place, mais essayez d'accéder au fichier `config/databases.yml` dans votre navigateur pour comprendre les conséquences négatives d'une telle attitude paresseuse. Si l'utilisateur sait que votre site est développé avec symfony, il aura accès à un grand nombre de fichiers sensibles.

N'utilisez jamais cette configuration sur un serveur de production, et lisez la section suivante pour savoir comment configurer votre serveur web correctement.

Configuration du serveur Web : La méthode sécurisée

Une bonne pratique web est de mettre sous le répertoire racine du site Web que les fichiers qui doivent être accessibles par un navigateur web, comme les feuilles de style, les Javascripts et les images. Par défaut, nous vous recommandons de stocker ces fichiers sous le répertoire `web/` du projet symfony et ses sous-répertoires.

Si vous jetez un œil sur ce répertoire, vous trouverez plusieurs sous-répertoires pour les ressources web (`css/` et `images/`) et deux fichiers de contrôleur frontal. Les contrôleurs frontaux sont seulement des fichiers PHP qui doivent être sous le répertoire racine web. Tous les autres fichiers PHP peuvent être cachés au navigateur, c'est une bonne idée en matière de sécurité.

Configuration du serveur Web

Maintenant il est temps de changer votre configuration d'Apache, pour rendre le nouveau projet accessible au monde.

Localisez et ouvrez le fichier de configuration `httpd.conf` et ajoutez la configuration suivante à la fin :

```
# Soyez sûr d'avoir seulement cette ligne une fois dans votre configuration
NameVirtualHost 127.0.0.1:8080

# C'est la configuration pour votre projet
Listen 127.0.0.1:8080

<VirtualHost 127.0.0.1:8080>
  DocumentRoot "/home/sfprojects/jobeet/web"
  DirectoryIndex index.php
  <Directory "/home/sfprojects/jobeet/web">
    AllowOverride All
    Allow from All
  </Directory>
```

```
  Alias /sf /home/sfprojects/jobeet/lib/vendor/symfony/data/web/sf
  <Directory "/home/sfprojects/jobeet/lib/vendor/symfony/data/web/sf">
    AllowOverride All
    Allow from All
  </Directory>
</VirtualHost>
```

L'alias `/sf` vous donne accès à des images et des fichiers JavaScript nécessaire pour afficher correctement les pages symfony par défaut et la barre d'outils web de débogage.

Sur Windows, vous devez remplacer la ligne `Alias` avec quelque chose comme :

```
Alias /sf "c:\dev\sfprojects\jobeet\lib\vendor\symfony\data\web\sf"
```

Et `/home/sfprojects/jobeet/web` doit être remplacé par :

```
c:\dev\sfprojects\jobeet\web
```

Cette configuration permet Apache d'écouter le port 8080 sur votre machine, de sorte que le site web sera accessible à l'adresse suivante :

```
http://~localhost~:8080/
```

Vous pouvez changer `8080` par un autre nombre, mais favorisez les nombres plus grand que `1024` car ils ne nécessitent pas de droits administrateurs.

Configurer un nom de domaine dédié

Si vous êtes un administrateur sur votre machine, il est préférable de configurer des serveurs virtuels plutôt que d'ajouter un nouveau port à chaque fois que vous démarrez un nouveau projet. Au lieu de choisir un port et d'ajouter une déclaration Listen, choisissez un nom de domaine (par exemple le nom du domaine réel avec `.localhost` ajouté à la fin) et ajouter une déclaration `ServerName` :

```
# C'est la configuration pour votre projet
<VirtualHost 127.0.0.1:80>
  ServerName www.jobeet.com.localhost
  <!-- same configuration as before -->
</VirtualHost>
```

Le nom du domaine `www.myproject.com.localhost` utilisé dans la configuration d'Apache doit être déclaré localement. Si vous utilisez un système Linux, il doit être fait dans le fichier `/etc/hosts`. Si vous exécutez Windows XP, ce fichier se trouve dans le répertoire `C:\WINDOWS\system32\drivers\etc\`.

Ajoutez la ligne suivante :

```
127.0.0.1 www.jobeet.com.localhost
```

Tester la nouvelle configuration

Redémarrez Apache, et vérifiez que vous avez maintenant accès à la nouvelle application en ouvrant un navigateur et en tapant `http://localhost:8080/index.php/`, ou `http://www.myproject.com.localhost/index.php/` en fonction de la configuration d'Apache que vous avez choisi dans la section précédente.

Si vous avez le module Apache `mod_rewrite` installé, vous pouvez retirer une partie de l'URL : `index.php/`. Ceci est possible grâce à la règle de reroutage configuré dans le fichier `web/.htaccess`.

Vous devriez également essayer d'accéder à l'application dans l'environnement de développement (voir la section suivante pour plus d'informations sur les environnements). Tapez l'adresse URL suivante :

```
http://www.jobeet.com.localhost/frontend_dev.php/
```

La barre d'outils web de débogage devrait apparaître dans le coin supérieur droit, incluant de petites icônes, prouvant que la configuration de votre alias `sf/` est correct.

La configuration est un peu différente si vous voulez faire tourner symfony sur un serveur IIS dans un environnement de Windows. Vous trouverez la façon de le configurer dans le tutoriel dédié.

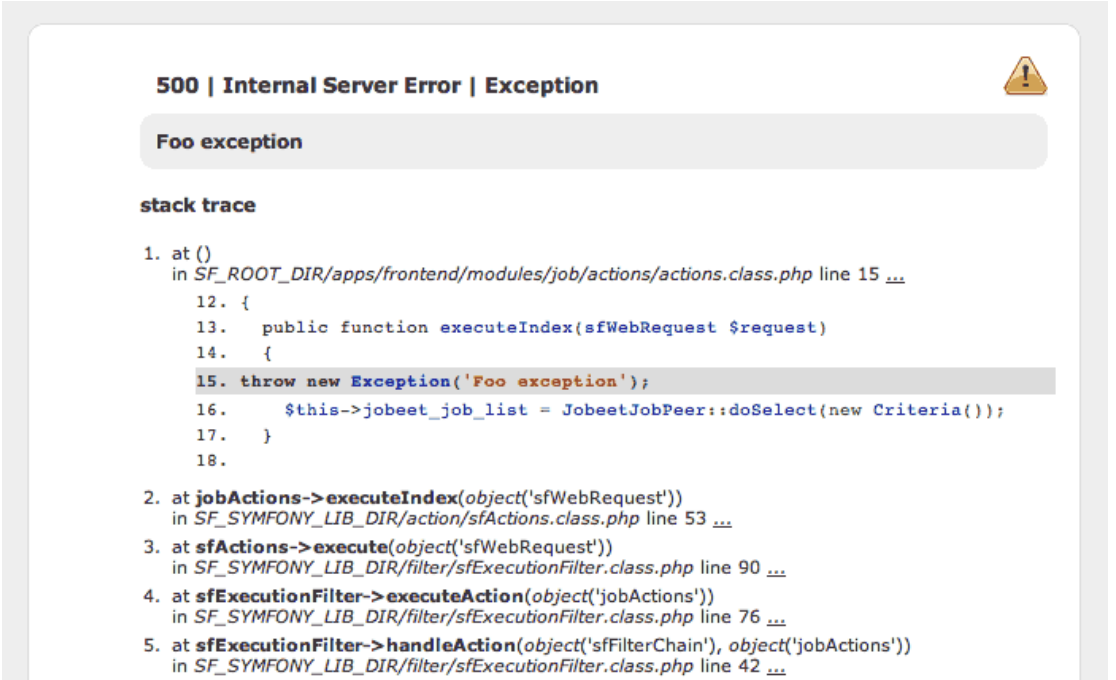
Les environnements

Si vous regardez le répertoire `web/`, vous trouverez deux fichiers PHP : `index.php` et `frontend_dev.php`. Ces fichiers sont appelés **contrôleurs frontaux**; toutes les requêtes de l'application se font par leur intermédiaire. Mais pourquoi nous avons deux contrôleurs frontaux pour chaque application ?

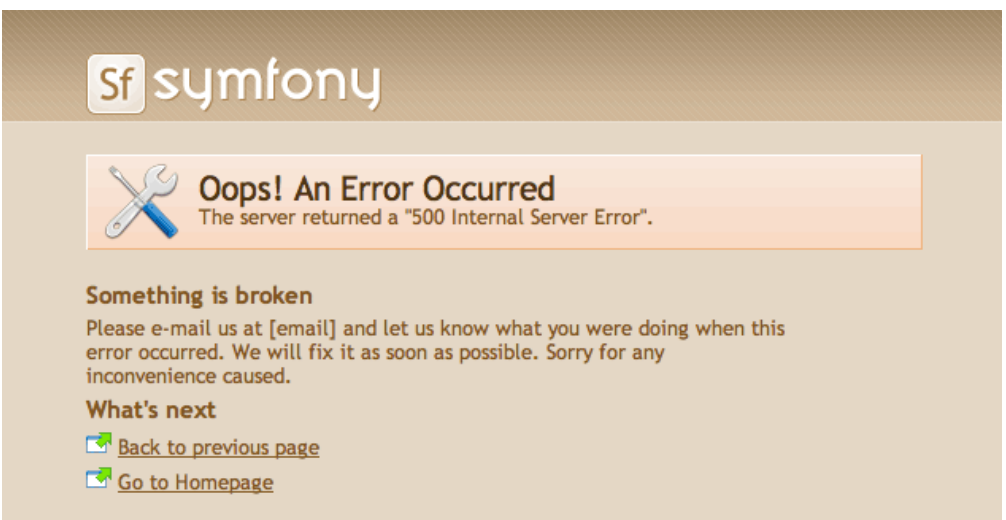
Les deux fichiers pointent sur la même application mais pour des **environnements** différents. Lorsque vous développez une application, sauf si vous développez directement sur le serveur de production, vous avez besoin de plusieurs environnements :

- L'**environnement de développement** : C'est l'environnement utilisé par les **développeurs Web** quand ils travaillent sur l'application pour ajouter de nouvelles fonctionnalités, corriger des bugs, ...
- L'**environnement de test** : Cet environnement est utilisé pour tester automatiquement l'application.
- L'**environnement de qualité** : Cet environnement est utilisé par le **client** pour tester l'application et les bogues ou les fonctionnalités manquantes.
- L'**environnement de production** : C'est l'environnement où interagissent les **utilisateurs finaux**

Qu'est ce qui rend un environnement unique ? Dans l'environnement de développement par exemple, l'application doit se connecter à tous les détails d'une requête afin de faciliter le débogage, mais le système de cache doit être désactivé de façon que tous les changements apportés au code soient pris en compte sans tarder. Ainsi, l'environnement de développement doit être optimisé pour le développeur. Le meilleur exemple est certainement lorsqu'une exception se produit. Pour aider le développeur à déboguer le problème plus rapidement, symfony affiche l'exception avec toutes les informations qu'elle a sur la requête courante dans le navigateur :



Par contre sur l'environnement de production, la couche du cache doit être activé, et bien entendu, l'application doit afficher les messages d'erreurs à la place des exceptions. Ainsi, l'environnement de production doit être optimisé pour la performance et l'expérience utilisateur.



Si vous ouvrez les fichiers des contrôleurs frontaux, vous verrez que leur contenu est le même, sauf pour la configuration de l'environnement :

```
// web/index.php
<?php

require_once(dirname(__FILE__).'/../config/ProjectConfiguration.class.php');

$configuration = ProjectConfiguration::getApplicationConfiguration('frontend',
'prod', false);
sfContext::createInstance($configuration)->dispatch();
```

La barre d'outils de débogage Web est également un excellent exemple de l'utilisation de l'environnement. Elle est présente sur toutes les pages dans l'environnement de développement et vous donne accès à un grand nombre d'informations en cliquant sur les différents onglets : la configuration de l'application actuelle, les journaux de la requête actuelle, les instructions SQL exécutées sur le moteur de base de données, des informations sur la mémoire, et l'heure.

Subversion

C'est une bonne pratique d'utiliser le contrôle de version des sources lors du développement d'une application web. En utilisant un contrôle de version des sources, cela nous permet de :

- travailler en toute confiance
- revenir à une version précédente si un changement casse quelquechose
- permettre à plusieurs personnes de travailler efficacement sur le projet
- avoir accès à toutes les versions successives de l'application

Dans cette section, nous décrirons comment utiliser Subversion avec symfony. Si vous utilisez un autre outil de contrôle de code source, il doit être assez facile d'adapter ce que nous décrivons pour Subversion.

Nous supposons que vous avez déjà accès à un serveur Subversion et que vous pouvez y accéder via HTTP.

Si vous ne disposez pas d'un serveur Subversion à votre disposition, vous pouvez créer un dépôt gratuit sur Google Code ou tapez simplement "free subversion repository" dans Google pour avoir beaucoup plus d'options.

Tout d'abord, créer un dépôt pour le projet jobeet sur le serveur de dépôt :

```
$ svnadmin create /path/to/jobeet/repository
```

Sur votre machine, créez la structure de répertoires de base :

```
$ svn mkdir -m "creer la structure de repertoires par defaut"
http://svn.example.com/jobeet/trunk http://svn.example.com/jobeet/tags
http://svn.example.com/jobeet/branches
```

Et faites un checkout du répertoire vide `trunk/` :

```
$ cd /home/sfprojects/jobeet
$ svn co http://svn.example.com/jobeet/trunk/ .
```

Ensuite, supprimez le contenu des répertoires `cache/` et `log/` car nous ne voulons pas les mettre dans le dépôt.

```
$ rm -rf cache/* log/*
```

Maintenant, assurez-vous de mettre les permissions d'écriture sur les répertoires du cache et des journaux aux niveaux appropriés afin que votre serveur web puisse écrire dedans :

```
$ chmod 777 cache/ log/
```

Maintenant, importer tous les fichiers et répertoires :

```
$ svn add *
```

Comme nous ne voudrons jamais faire de commit des fichiers situés dans les répertoires `cache/` et `log/`, vous devez spécifier une liste à ignorer :

```
$ svn propedit svn:ignore cache
```

L'éditeur de texte par défaut configuré dans SVN devrait se lancer. Subversion doit ignorer tout le contenu de ce répertoire :

```
*
```

Sauvegardez et quittez. Vous avez terminé.

Répétez la procédure pour le répertoire `log/` :

```
$ svn propedit svn:ignore log
```

Et entrez :

```
*
```

Enfin, valider ces modifications dans le dépôt :

```
$ svn import -m "fait l import initial" . http://svn.example.com/jobeet/trunk
```

Les utilisateurs de Windows peuvent utiliser l'excellent client TortoiseSVN pour gérer leur

dépôt Subversion.

Conclusion

Ce premier chapitre s'achève ici. Bien que nous n'ayons pas encore commencé à parler de symfony, nous avons cependant créé un environnement de développement solide. Nous avons aussi évoqué quelques bonnes pratiques du développement web, et nous sommes à présent prêts à développer.

Le chapitre suivant dévoilera les fonctionnalités de la future application et détaillera les besoins fonctionnels et techniques à satisfaire dans Jobeet.

Jour 2 : Le Projet »

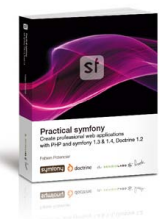
Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 2 : Le Projet

Nous n'avons pas écrit une seule ligne de PHP pour le moment, mais hier, nous avons configuré l'environnement, créé un projet symfony vide, et fait en sorte que nous commencions avec une bonne sécurité par défaut. Si vous avez suivi tout le long, vous avez regardé votre écran avec délice depuis qu'il affiche la belle page par défaut de symfony pour de nouvelles applications.



Mais vous en voulez plus. Vous voulez apprendre les bases et l'essentiel des détails du développement d'une application symfony. Donc, reprenons notre voyage pour atteindre le nirvana du développement de symfony.

Aujourd'hui, nous allons prendre le temps de décrire les exigences du projet Jobeet avec quelques maquettes de base.

Le pitch du projet

Tout le monde parle de la crise de nos jours. Le chômage augmente à nouveau.

Je sais, les développeurs symfony ne sont pas trop concernés et c'est pourquoi vous voulez apprendre symfony en premier lieu. C'est également difficile de trouver de bons développeurs symfony.

Où pouvez-vous trouver un développeur symfony ? Où pouvez-vous montrer vos compétences en symfony ?

Vous devez trouvez un bon site pour la recherche d'emploi. Vous pensez à Monster ? Réfléchissez encore. Vous avez besoin d'un site dédié. Un site où vous pourrez trouver les meilleures personnes, des experts. Un site où il est simple, rapide, et fun de rechercher un travail, ou d'en proposer.

Ne cherchez plus. Jobeet est le site. **Jobeet est un logiciel open-source de recherche d'emploi** qui ne fait qu'une seule chose, mais le fait bien. Il est facile d'utilisation, à adapter, à faire évoluer, et à intégrer à votre site internet. Il est multi langues dès le départ, et bien sûr il utilise les dernières technologies du web 2.0 pour améliorer l'expérience utilisateur. Il fournit également des feeds et une API pour interagir avec lui en programmant.

Est-ce qu'il existe déjà ? Comme utilisateur, vous trouverez beaucoup de site de recherche d'emplois comme Jobeet sur internet. Mais trouvez-en un qui est Open Source, et qui possède des fonctionnalités aussi évoluées que celles que nous vous proposons.

Si vous cherchez un emploi sur symfony ou que vous voulez faire appel à un développeur symfony, vous pouvez aller sur le site symfonians.

Les histoires d'utilisateur du projet

Avant de plonger la tête la première dans le code, nous allons décrire le projet un peu plus. Les sections suivantes décrivent les fonctionnalités que nous voulons mettre en œuvre dans la première version/itération du projet avec des histoires simples.

Le site web Jobeet a quatre types d'utilisateurs :

- **L'administrateur** : il est propriétaire du site et il a le pouvoir magique
- **L'utilisateur** : il visite le site pour chercher un emploi
- **L'annonceur** : il soumet une offre d'emploi
- **L'affilié** : il re-édite certains emplois sur son site

Le projet a deux applications : le **frontend** (les histoires F1 à F7, ci-dessous), où les utilisateurs interagissent avec le site, et le **backend** (les histoires B1 à B3), où les administrateurs gèrent le site.

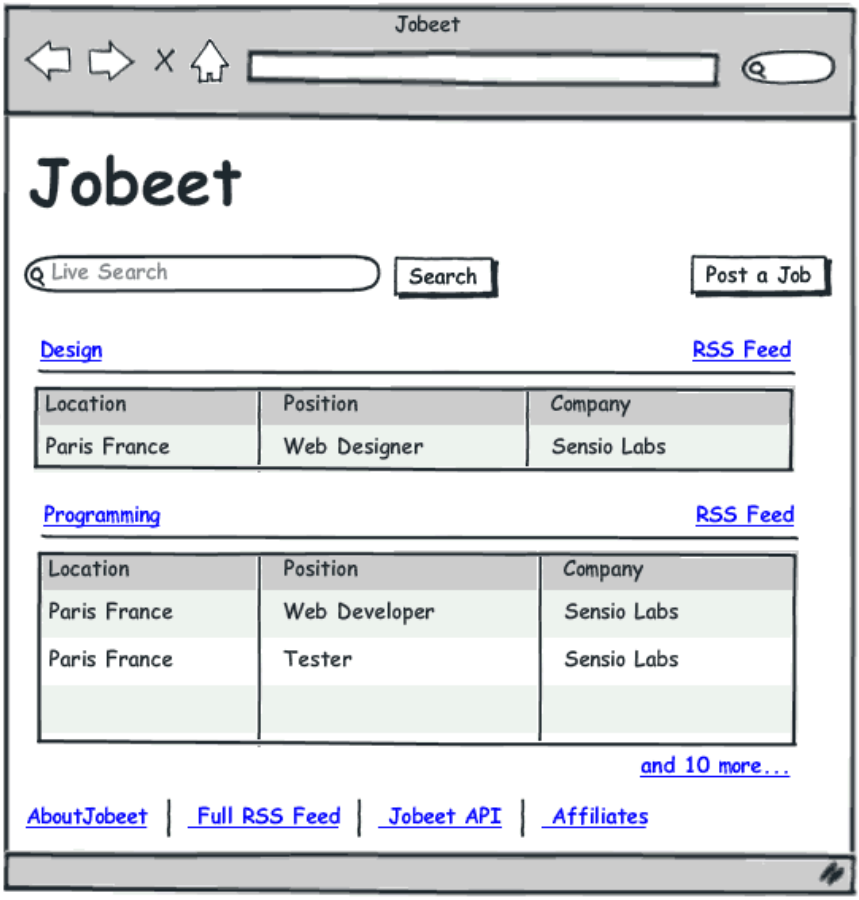
L'application backend est sécurisé et requiert des droits d'accès.

Histoire F1 : Sur la page d'accueil, l'utilisateur voit les dernières offres actives

Quand un utilisateur vient sur le site Jobeet, il voit une liste des emplois actifs. Les emplois sont classés par catégorie, puis par date de publication (emplois plus récents en premier). Pour chaque emploi, seul le lieu, la position, et la société sont affichées.

Pour chaque catégorie, la liste ne montre que les 10 premiers emplois et un lien permet de lister tous les emplois pour une catégorie donnée (Histoire F2).

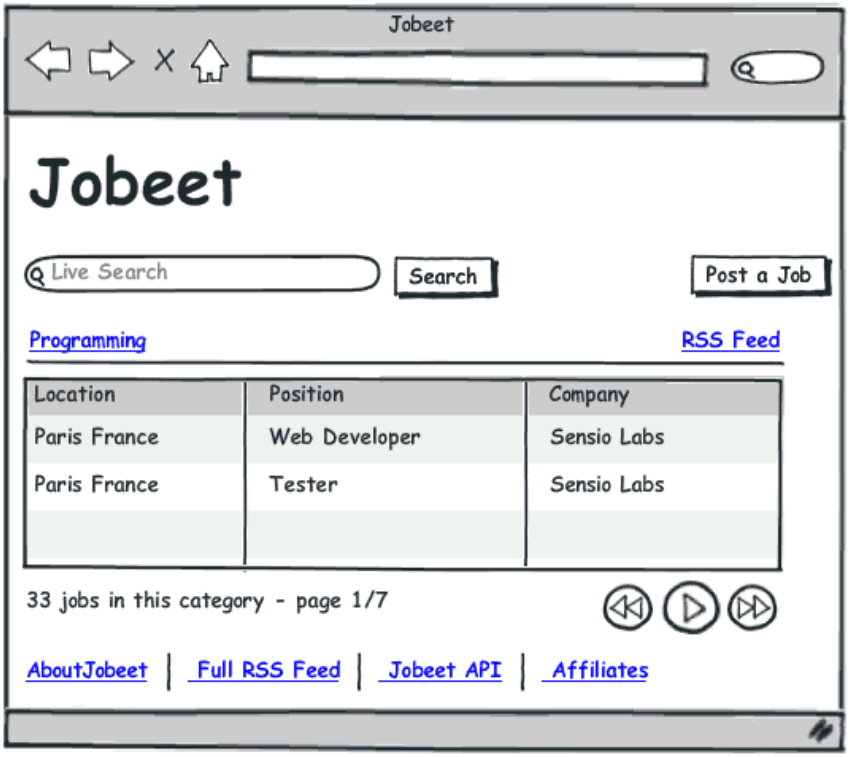
Sur la page d'accueil , l'utilisateur peut affiner la liste des travaux (*Histoire F3*), ou par la soumission d'un nouvel emploi (Histoire F5).



Histoire F2 : Un utilisateur peut demander tous les emplois dans une catégorie donnée

Quand un utilisateur clique sur un nom de catégorie ou sur le lien "more jobs" sur la page d'accueil, il voit tous les emplois pour cette catégorie triée par date.

La liste est paginée avec 20 emplois par page.

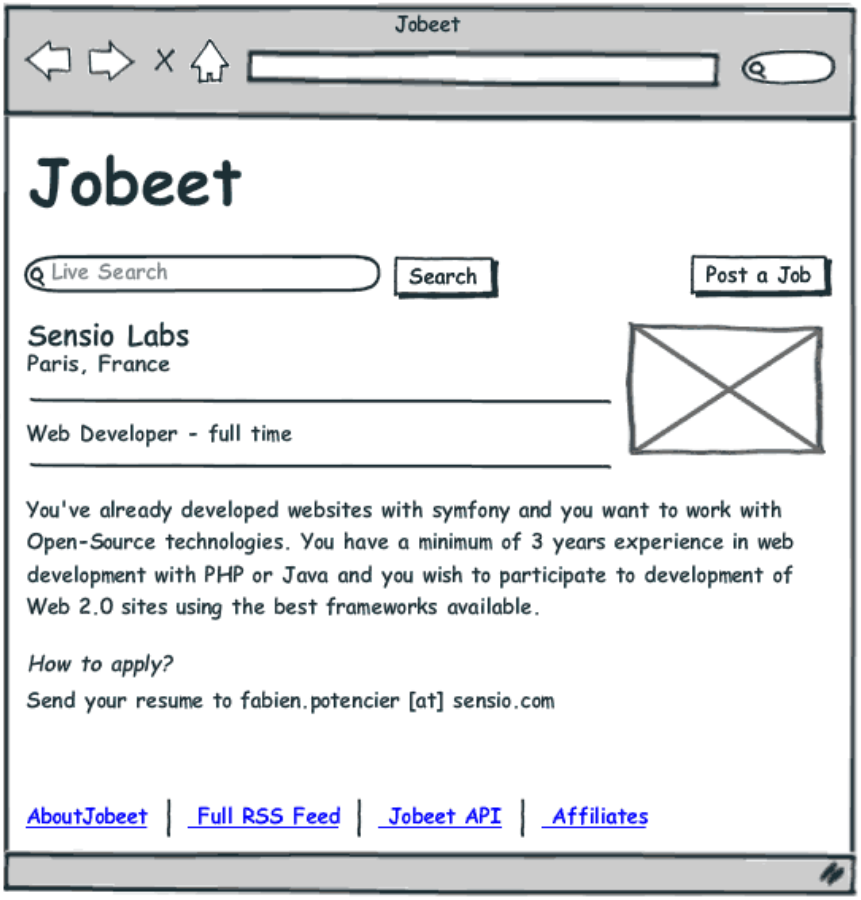


Histoire F3 : Un utilisateur affine la liste avec quelques mots clés

L'utilisateur peut saisir quelques mots clés pour affiner sa recherche. Les mots clés peuvent être des mots trouvés dans l'emplacement, la position, la catégorie, ou les champs de l'entreprise.

Histoire F4 : Un utilisateur clique sur un emploi pour obtenir des informations plus détaillées

L'utilisateur peut sélectionner un emploi dans la liste pour afficher des informations plus détaillées.



Histoire F5 : Un utilisateur soumet un emploi

Un utilisateur peut soumettre un emploi. Un emploi est composé de plusieurs l'informations :

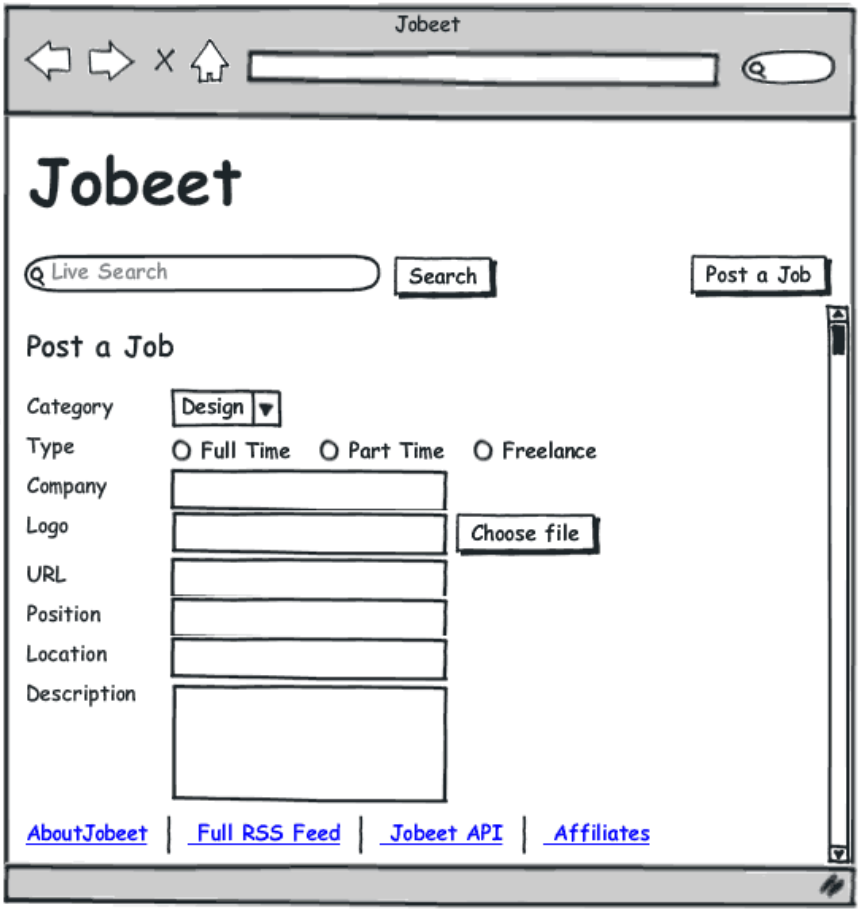
- Société
- Type (à temps plein, à temps partiel, ou freelance)
- Logo (facultatif)
- URL (facultatif)
- Poste
- Localité
- Catégorie (l'utilisateur choisit dans une liste de catégories possibles)
- Description du poste (les URLs et les emails sont automatiquement mis en lien)
- Comment appliquer (les URL et les emails sont automatiquement mis en lien)
- Publique (si la tâche peut également être publiés sur les sites affiliés)
- Email (l'email de celui qui a soumis l'emploi)

Il n'est pas nécessaire de créer un compte afin de soumettre un emploi.

Le processus est simple, avec seulement deux étapes : d'abord, l'utilisateur remplit dans le formulaire toutes les informations nécessaires pour décrire l'emploi, puis il valide les informations en visualisant la page finale de l'emploi.

Même si l'utilisateur n'a pas de compte, un emploi peut être modifié ultérieurement, grâce à une URL spécifique (protégé par un jeton donné à l'utilisateur lorsque l'emploi est créé).

Chaque poste de travail est en ligne pendant 30 jours (ce qui est configurable par l'administrateur - voir *Histoire B2*). Un utilisateur peut revenir réactiver ou prolonger la validité de l'annonce pour un supplément de 30 jours, mais seulement lorsque le travail expire dans moins de 5 jours.



Histoire F6 : Un utilisateur demande à devenir une société affiliée

Un utilisateur doit demander à devenir un affilié et être autorisés à utiliser l'API Jobeet. Pour postuler, il doit donner les informations suivantes :

- Nom
- Email
- URL du site web

Le compte d' un affilié doit être activé par l'administrateur (*Histoire B3*). Une fois activé, l'affilié reçoit un jeton pour une utilisation avec l'API par email.

Lors de l'application, l'affilié peut également choisir d'obtenir des emplois auprès d'un sous-ensemble des catégories disponibles.

Histoire F7 : Un affilié récupère la liste actuelle des emplois actifs

Un affilié peut récupérer la liste des emplois en cours en appelant l'API avec le jeton de sa filiale. La liste peut être retournée en XML, JSON ou YAML.

La liste contient les informations publiques disponibles pour un emploi.

L'affilié peut également limiter le nombre d'emplois qui seront retournés, et d'affiner sa requête en spécifiant une catégorie.

Histoire B1 : Un administrateur configure le site web

L'administrateur peut modifier les catégories disponibles sur le site.

Histoire B2 : Un administrateur gère les emplois

Un administrateur peut modifier et supprimer tous les emplois affichés.

Histoire B3 : Un administrateur gère les affiliés

L'administrateur peut créer ou modifier des affiliés. Il est responsable de l'activation d'un affilié et peut également les désactiver.

Lorsque l'administrateur active une nouvelle filiale, le système crée un jeton unique à utiliser par la filiale.

Conclusion

Comme pour tout développement Web, vous ne commencez jamais la programmation la première journée. Vous avez besoin de recueillir les premières exigences et les travaux sur une maquette. C'est ce que nous avons fait aujourd'hui.

« Jour 1 : Démarrage du projet

Jour 3 : Le modèle de données »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

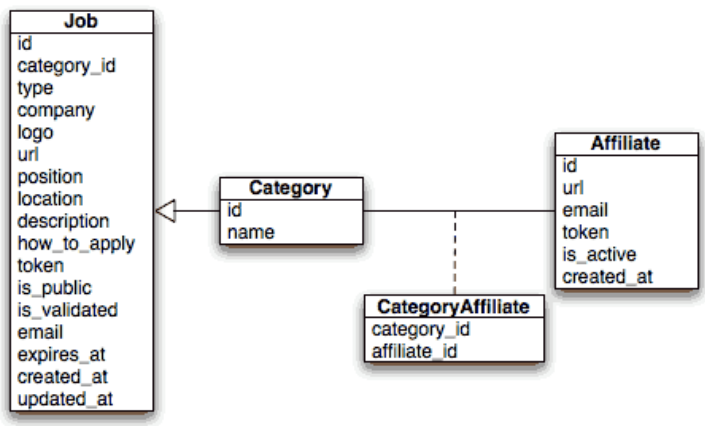
If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 3 : Le modèle de données

Ceux d'entre vous qui brûlent d'ouvrir leur éditeur de texte et de définir un peu de PHP seront heureux d'apprendre que le tutoriel d'aujourd'hui va nous entraîner dans un certain développement. Nous allons définir le modèle de données de Jobeet, utiliser un ORM pour interagir avec la base de données, et construire le premier module de l'application. Mais, comme symfony fait beaucoup de travail pour nous, nous aurons un module web pleinement opérationnel sans trop écrire de code PHP.

Le modèle relationnel

Les histoires d'utilisateurs que nous avons écrites hier, décrivent les objets principaux de notre projet : les emplois, les sociétés affiliées et les catégories. Voici le schéma correspondant aux relations entre entités :



En plus des colonnes décrites dans les histoires, nous avons également ajouté un champ `created_at` à certaines tables. Symfony reconnaît de tels champs et définit la valeur avec celle de l'heure actuelle du système quand un enregistrement est créé. C'est la même chose pour les champs `updated_at` : leur valeur est définie avec celle de l'heure du système quand l'enregistrement est mis à jour.

Le schéma

Pour stocker les emplois, les sociétés affiliées et les catégories, il nous faut évidemment une base de données relationnelle.

Mais comme Symfony est un framework orienté-objet, on aime manipuler des objets quand nous le pouvons. Par exemple, au lieu d'écrire des instructions SQL pour récupérer des enregistrements de la base de données, nous préférons utiliser des objets.

Les informations de base de données relationnelle doivent être mappées sur un modèle objet. Cela peut être fait avec un outil ORM et heureusement, symfony est livré avec deux d'entre eux : Propel et Doctrine. Dans ce tutoriel, nous allons utiliser Doctrine.

L'ORM a besoin d'une description des tables et leurs relations pour créer les classes liées. Il y a deux façons de créer ce schéma de description : par introspection d'une base de données existante ou en la créant à la main.

Comme la base de données n'existe pas encore et que nous voulons garder agnostique la base de données Jobeet, nous allons créer le fichier du schéma à la main en éditant le fichier vide `config/doctrine/schema.yml` :

```
# config/doctrine/schema.yml
JobeetCategory:
```

```
  actAs: { Timestampable: ~ }
  columns:
    name: { type: string(255), notnull: true, unique: true }

JobeetJob:
  actAs: { Timestampable: ~ }
  columns:
    category_id:  { type: integer, notnull: true }
    type:         { type: string(255) }
    company:      { type: string(255), notnull: true }
    logo:         { type: string(255) }
    url:          { type: string(255) }
    position:     { type: string(255), notnull: true }
    location:     { type: string(255), notnull: true }
    description:  { type: string(4000), notnull: true }
    how_to_apply: { type: string(4000), notnull: true }
    token:        { type: string(255), notnull: true, unique: true }
    is_public:    { type: boolean, notnull: true, default: 1 }
    is_activated: { type: boolean, notnull: true, default: 0 }
    email:        { type: string(255), notnull: true }
    expires_at:   { type: timestamp, notnull: true }
  relations:
    JobeetCategory: { onDelete: CASCADE, local: category_id, foreign: id,
foreignAlias: JobeetJobs }

JobeetAffiliate:
  actAs: { Timestampable: ~ }
  columns:
    url:       { type: string(255), notnull: true }
    email:     { type: string(255), notnull: true, unique: true }
    token:     { type: string(255), notnull: true }
    is_active: { type: boolean, notnull: true, default: 0 }
  relations:
    JobeetCategories:
      class: JobeetCategory
      refClass: JobeetCategoryAffiliate
      local: affiliate_id
      foreign: category_id
      foreignAlias: JobeetAffiliates

JobeetCategoryAffiliate:
  columns:
    category_id:  { type: integer, primary: true }
    affiliate_id: { type: integer, primary: true }
  relations:
    JobeetCategory:  { onDelete: CASCADE, local: category_id, foreign: id }
    JobeetAffiliate: { onDelete: CASCADE, local: affiliate_id, foreign: id }
```

Si vous avez décidé de créer les tables en écrivant des instructions SQL, vous pouvez générer le fichier de configuration correspondant `schema.yml` en exécutant la tâche `doctrine:build-schema` :

```
$ php symfony doctrine:build-schema
```

La tâche ci-dessus suppose que vous ayez une base de données configurées dans `databases.yml`. Nous vous montrerons comment configurer la base de données dans une étape suivante. Si vous essayez et exécutez cette tâche maintenant, elle ne fonctionnera pas car elle ne connait pas la base de données pour construire le schéma.

Le schéma est la traduction directe du diagramme de relations des entités dans le format YAML.

Le format YAML

Selon le site officiel YAML, YAML est "un standard de sérialisation de données compréhensibles par un humain, quel que soit le langage de programmation".

Autrement dit, YAML est un langage simple pour décrire les données (des chaînes, des entiers, des dates, des tableaux et des séries).

En YAML, la structure est présentée à travers l'indentation, la séquence des éléments sont

indiqués par un tiret, et les paires clé/valeur dans le fichier sont séparés par deux points. YAML possède également une syntaxe abrégée pour décrire la même structure avec moins de lignes, où les tableaux sont explicitement indiqués avec `[]` et les séries avec `{}`.

Si vous n'êtes pas encore familiarisé avec YAML, il est temps de commencer car le framework symfony l'utilise intensivement pour ses fichiers de configuration. Un bon point de départ est la partie de la documentation de symfony pour YAML.

Il y a une chose importante que vous devez retenir lors de l'édition d'un fichier YAML : **l'indentation doit être faite avec un ou plusieurs espaces, mais jamais avec les tabulations**.

Le fichier `schema.yml` contient la description de toutes les tables et leurs colonnes. Chaque colonne est décrite avec les informations suivantes :

- `type`: Le type de colonne (`boolean`, `integer`, `float`, `decimal`, `string`, `array`, `object`, `blob`, `clob`, `timestamp`, `time`, `date`, `enum`, `gzip`)
- `notnull`: Réglez-le à `true` si vous souhaitez que la colonne soit requise
- `unique`: Réglez-le à `true` si vous souhaitez créer un index unique pour la colonne.

L'attribut `onDelete` définit le comportement `ON DELETE` des clés étrangères, et Doctrine supporte `CASCADE`, `SETNULL`, et `RESTRICT`. Par exemple, quand un enregistrement `job` est supprimé, tous les enregistrements connexes `jobeet_category_affiliate` seront automatiquement supprimés par la base de données.

La base de données

Le framework symfony supporte toutes les PDO-supporté par les bases de données (MySQL, PostgreSQL, SQLite, Oracle, MSSQL, ...). PDO est la couche d'abstraction de la base de données intégrée à PHP.

Utilisons MySQL pour ce tutoriel:

```
$ mysqladmin -uroot -p create jobeet
Enter password: mYsEcret ## The password will echo as ********
```

N'hésitez pas à choisir un autre moteur de base de données si vous le voulez. Il ne sera pas difficile d'adapter le code que nous allons écrire car nous allons utiliser l'ORM qui va écrire le code SQL pour nous.

Nous devons dire à symfony d'utiliser cette base de données pour le projet Jobeet :

```
$ php symfony configure:database "mysql:host=localhost;dbname=jobeet" root mYsEcret
```

La tâche `configure:database` prend trois arguments: le PDO DSN, le nom d'utilisateur et le mot de passe pour accéder à la base de données. Si vous n'avez pas besoin d'un mot de passe pour accéder à votre base de données sur le serveur de développement, omettez simplement le troisième argument.

La tâche `configure:database` stocke la configuration de la base de données dans le fichier de configuration de `config/databases.yml`. Au lieu d'utiliser la tâche, vous pouvez éditer ce fichier à la main.

En passant le mot de passe base de données sur la ligne de commande est pratique mais peu sûr. En fonction de qui a accès à votre environnement, il pourrait être préférable de modifier le fichier `config/databases.yml` pour changer le mot de passe. Bien sûr, pour garder le mot de passe sûr, le mode d'accès du fichier de configuration devraient également être limités.

L'ORM

Grâce à la description de base de données à partir du fichier `schema.yml`, nous pouvons utiliser les tâches intégrées Doctrine qui génèrent les instructions SQL nécessaires pour créer les tables de la base de données :

D'abord afin de générer le code SQL, vous devez construire vos modèles à partir de vos fichiers du schéma.

```
$ php symfony doctrine:build --model
```

Maintenant que vos modèles sont présents, vous pouvez générer et insérer le code SQL.

```
$ php symfony doctrine:build --sql
```

La tâche `doctrine:build --sql` génère les instructions SQL dans le répertoire `data/sql/`, optimisées pour le moteur de la base de données que nous avons configuré :

```
# snippet from data/sql/schema.sql
CREATE TABLE jobeet_category (id BIGINT AUTO_INCREMENT, name VARCHAR(255)
NOT NULL COMMENT 'test', created_at DATETIME, updated_at DATETIME, slug
VARCHAR(255), UNIQUE INDEX sluggable_idx (slug), PRIMARY KEY(id))
ENGINE = INNODB;
```

Pour créer les tables dans la base de données, vous devez exécuter la tâche `doctrine:insert-sql` :

```
$ php symfony doctrine:insert-sql
```

Comme pour tout outil en ligne de commande, les tâches de symfony peuvent prendre des arguments et des options. Chaque tâche est livrée avec un message d'aide intégré qui peut être affiché en exécutant la tâche `help` :

```
$ php symfony help doctrine:insert-sql
```

Le message d'aide liste tous les arguments et les options possibles, il donne les valeurs par défaut pour chacune d'elles, et fournit quelques exemples d'utilisation utile.

L'ORM génère également des classes PHP qui met les enregistrements de la table en objets :

```
$ php symfony doctrine:build --model
```

La tâche `doctrine:build --model` crée des fichiers PHP dans le répertoire `lib/model/` qui peut être utilisé pour interagir avec la base de données.

En naviguant dans les fichiers générés, vous avez probablement remarqué que Doctrine génère trois classes par table. Pour la table `jobeet_job` :

- `JobeetJob`: Un objet de cette classe représente un seul enregistrement de la table de `jobeet_job`. La classe est vide par défaut.
- `BaseJobeetJob`: La classe parent de `JobeetJob`. Chaque fois que vous exécutez `doctrine:build --model`, cette classe est écrasée, ainsi toutes les personnalisations doivent être faites dans la classe `JobeetJob`.
- `JobeetJobTable`: La classe définit des méthodes qui renvoient surtout des collections d'objets de `JobeetJob`. La classe est vide par défaut.

Les valeurs des colonnes d'un enregistrement peuvent être manipulées avec un objet modèle en utilisant des accesseurs (des méthodes get*()) et des mutateurs (des méthodes set*()) :

```
$job = new JobeetJob();
$job->setPosition('Web developer');
$job->save();

echo $job->getPosition();

$job->delete();
```

Vous pouvez également définir des clés étrangères en reliant directement les objets entre eux :

```
$category = new JobeetCategory();
$category->setName('Programming');

$job = new JobeetJob();
$job->setCategory($category);
```

La tâche `doctrine:build --all` est un raccourci des tâches que nous avons exécuté dans cette section et plus encore. Aussi, exécutez cette tâche maintenant pour génèrer les formulaires et les validateurs pour les classes modèle de Jobeet :

```
$ php symfony doctrine:build --all --no-confirmation
```

Vous pourrez voir les validateurs à l'oeuvre à la fin de la journée et les formulaires seront bien expliquées en détail le jour 10.

Les données initiales

Les tables ont été créées dans la base de données mais elles n'ont pas de données. Pour toute application web, il existe trois types de données :

- **Les données initiales** : Les données initiales sont nécessaires à l'application pour travailler. Par exemple, Jobeet a besoin de quelques catégories initiales. Sinon, personne ne sera en mesure de soumettre un emploi. Nous avons également besoin d'un utilisateur administrateur pour être capable de se connecter au backend.
- **Les données de test** : Les données de test sont nécessaires pour l'application à tester. En tant que développeur, vous devrez écrire des tests pour s'assurer que Jobeet se comporte comme décrit dans les histoires d'utilisateur, et la meilleure façon est d'écrire des tests automatisés. Donc, chaque fois que vous exécutez vos tests, vous avez besoin d'une base de données propre avec quelques nouvelles données pour tester tout de suite.
- **Les données utilisateurs** : Les données utilisateurs sont créés par des utilisateurs pendant la durée de vie normale de l'application.

Chaque fois que symfony crée les tables dans la base de données, toutes les données sont perdues. Pour peupler la base de données avec des données initiales, nous pourrions créer un script PHP, ou exécuter des instructions SQL avec le programme `mysql`. Mais comme le besoin est assez fréquent, il y a une meilleure façon avec symfony : créer des fichiers YAML dans le répertoire `data/fixtures/` et utiliser la tâche `doctrine:data-load` pour les charger dans la base de données.

Premièrement, créer les fichiers de jeu de test suivants :

```
# data/fixtures/categories.yml
JobeetCategory:
  design:
    name: Design
  programming:
    name: Programming
  manager:
    name: Manager
  administrator:
    name: Administrator

# data/fixtures/jobs.yml
JobeetJob:
  job_sensio_labs:
    JobeetCategory: programming
    type:         full-time
    company:      Sensio Labs
    logo:         sensio-labs.gif
    url:          http://www.sensiolabs.com/
    position:     Web Developer
    location:     Paris, France
    description:  |
      You've already developed websites with symfony and you want to work
      with Open-Source technologies. You have a minimum of 3 years
      experience in web development with PHP or Java and you wish to
      participate to development of Web 2.0 sites using the best
      frameworks available.
    how_to_apply: |
      Send your resume to fabien.potencier [at] sensio.com
    is_public:    true
    is_activated: true
    token:        job_sensio_labs
    email:        job@example.com
    expires_at:   '2010-10-10'

  job_extreme_sensio:
    JobeetCategory:  design
    type:         part-time
    company:      Extreme Sensio
```

```
    logo:         extreme-sensio.gif
    url:          http://www.extreme-sensio.com/
    position:     Web Designer
    location:     Paris, France
    description:  |
      Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
      eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
      enim ad minim veniam, quis nostrud exercitation ullamco laboris
      nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
      in reprehenderit in.

      Voluptate velit esse cillum dolore eu fugiat nulla pariatur.
      Excepteur sint occaecat cupidatat non proident, sunt in culpa
      qui officia deserunt mollit anim id est laborum.
    how_to_apply: |
      Send your resume to fabien.potencier [at] sensio.com
    is_public:    true
    is_activated: true
    token:        job_extreme_sensio
    email:        job@example.com
    expires_at:   '2010-10-10'
```

Le fichier de jeu de test des emplois fait référence à deux images. Vous pouvez les télécharger (http://www.symfony-project.org/get/jobeet/sensio-labs.gif, http://www.symfony-project.org/get/jobeet/extreme-sensio.gif) et les mettre sous le répertoire `web/uploads/jobs/`.

Un fichier de jeu de test est écrit en YAML, et définit les objets du modèle, étiquetés avec un nom unique (par exemple, nous avons défini deux emplois étiquetés `job_sensio_labs` et `job_extreme_sensio`). Cette étiquette est d'une grande utilitée pour relier les objets liés, sans avoir à définir les clés primaires (qui sont souvent auto-incrémenté et ne peuvent pas être attribuées). Par exemple, la catégorie d'emploi de `job_sensio_labs` est `programming`, qui est l'étiquette donnée à la catégorie 'Programming'.

Dans un fichier YAML, quand une chaîne contient des sauts de ligne (comme la colonne `description` dans le fichier de jeu de test d'emploi), vous pouvez utiliser le pipe (`|`) pour indiquer que la chaîne va continuer sur plusieurs lignes.

Bien qu'un fichier de jeu de test peut contenir des objets d'un ou de plusieurs modèles, nous avons décidé de créer un fichier par modèle pour les jeux de test de Jobeet.

Propel exige que les fichiers de jeux de test soient préfixés avec un nombre pour déterminer l'ordre dans lequel les fichiers seront chargés. Avec Doctrine, ce n'est pas nécessaire car tous les jeux de test seront chargés et enregistrés dans le bon ordre afin de s'assurer que les clés étrangères soient définies correctement.

Dans un fichier de jeu de test, vous n'avez pas besoin de définir toutes les valeurs des colonnes. Sinon, symfony va utiliser la valeur par défaut définie dans le schéma de base de données. Et comme symfony utilise Doctrine pour charger les données dans la base de données, tous les comportements intégrés (comme le paramètrage automatique des colonnes `created_at` ou `updated_at`) et les comportements personnalisés que vous pourrez ajouter dans les classes du modèle qui sont activés.

Le chargement des données initiales dans la base de données est aussi simple que de lancer la tâche `doctrine:data-load` :

```
$ php symfony doctrine:data-load
```

La tâche `doctrine:build --all --and-load` est un raccourci pour la tâche `doctrine:build --all` suivie par la tâche `doctrine:data-load`.

Exécutez la tâche `doctrine:build --all --and-load` pour s'assurer que tout est généré à partir de votre schéma. Cela va générer vos formulaires, vos filtres, vos modèles, supprimer votre base de données et re-créer toutes les tables.

```
$ php symfony doctrine:build --all --and-load
```

Le voir en action dans le navigateur

Nous avons beaucoup utilisé l'interface en ligne de commande, mais ce n'est pas vraiment excitant, surtout pour un projet web. Nous avons maintenant tout ce qu'il faut pour créer des pages web qui interagissent avec la base de données.

Voyons comment afficher la liste des emplois, comment modifier un emploi existant et la façon de supprimer un emploi. Comme expliqué au cours du jour 1, un projet symfony est composé d'applications. Chaque application est ensuite divisé en **modules**. Un module est un ensemble autonome de code PHP qui représente une caractéristique de l'application (le module API par exemple), ou un ensemble de manipulations, l'utilisateur peut le faire sur un modèle d'objet (un module emploi par exemple).

Symfony est capable de générer automatiquement un module pour un modèle donné qui fournit des fonctions de manipulation de base :

```
$ php symfony doctrine:generate-module --with-show --non-verbose-templates frontend
job JobeetJob
```

La `doctrine:generate-module` génère un module `job` dans l'application `frontend` pour le modèle `JobeetJob`. Comme pour la plupart des tâches de symfony, certains fichiers et certains répertoires ont été créés pour vous sous le répertoire `apps/frontend/modules/job/` :

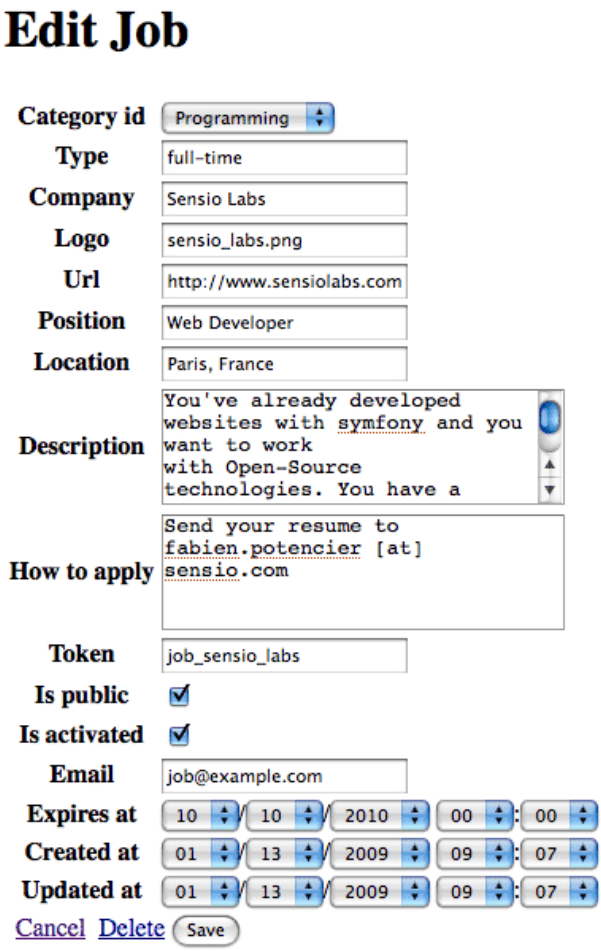
Répertoire	Description
`actions/`	Les actions du module
`templates/`	Les Templates du module

The `actions/actions.class.php` file defines all the available **action** for the `job` module:

Nom de l'action	Description
`index`	Affiche les enregistrements de la table
`show`	Affiche les champs et leurs valeurs pour un enregistrement donné
`new`	Affiche le formulaire pour créer un nouvel enregistrement
`create`	Crée un nouvel enregistrement
`edit`	Affiche le formulaire pour éditer un enregistrement existant
`update`	Met à jour un enregistrement en fonction des valeurs sousmises par l'utilisateur
`delete`	Supprime un enregistrement donné de la table

Vous pouvez tester le module job dans un navigateur :

```
http://www.jobeet.com.localhost/frontend_dev.php/job
```

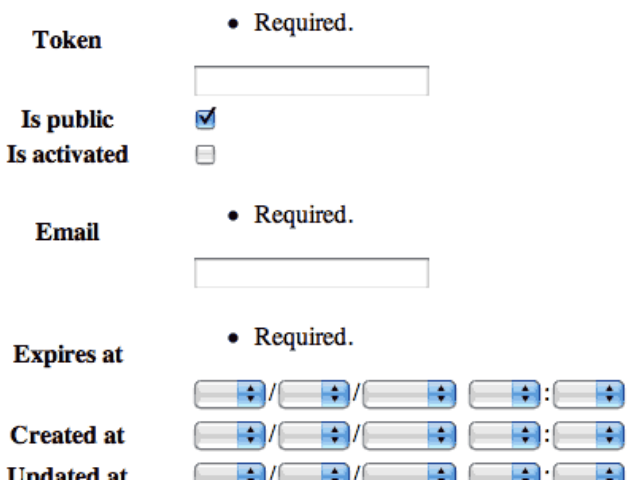


Si vous essayez de modifier un emploi, vous remarquerez que l'id de la catégorie a une liste de tous les noms des catégories. La valeur de chaque option est obtenu à partir de la méthode `__toString()`.

Doctrine va essayer de fournir une méthode de base `__toString()` en devinant un nom de colonne descriptif comme `title`, `name`, `subject`, etc. Si vous voulez quelque chose de personnalisé, vous aurez besoin d'ajouter vos propres méthodes `__toString()` comme ci-dessous. Le modèle `JobeetCategory` est capable de deviner la méthode `__toString()` en utilisant le nom de la colonne de la table `jobeet_category`.

```
// lib/model/doctrine/JobeetJob.class.php
class JobeetJob extends BaseJobeetJob
{
  public function __toString()
  {
    return sprintf('%s at %s (%s)', $this->getPosition(), $this->getCompany(), $this-
>getLocation());
  }
}

// lib/model/doctrine/JobeetAffiliate.class.php
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function __toString()
  {
    return $this->getUrl();
  }
}
```

Vous pouvez désormais créer et éditer des emplois. Essayez de laisser un champ obligatoire vide, ou essayez d'entrer une date invalide. C'est vrai, symfony a créé des règles de validation de base par introspection du schéma de la base de données.



Conclusion

C'est tout pour ce chapitre. Nous vous avions prévenu en guise introduction. Au cours de ce troisième chapitre, nous avons à peine écrit du code PHP, mais nous avons néanmoins un module web de travail pour le modèle job, prêt à être modifié et personnalisé. Rappelez-vous, pas de code PHP signifie pas de bogues non plus !

Si vous avez encore de l'énergie, n'hésitez pas à lire le code généré pour le module et le modèle et essayer de comprendre comment il fonctionne. Sinon, ne vous inquiétez pas et dormez bien, car demain nous allons parler de l'un des paradigmes le plus utilisé dans les frameworks web, le modèle de conception MVC.

« Jour 2 : Le Projet

Jour 4 : Le contrôleur et la vue »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 4 : Le contrôleur et la vue

Hier, nous avons appris comment symfony nous simplifie la gestion des bases de données en faisant abstraction des différences entre les moteurs de base de données et en convertissant les éléments relationnels sous forme de classes orientées objets. Nous avons également vu le principe de fonctionnement de Doctrine permettant de définir le schéma de la base de données, créer les tables et remplir la base de données avec quelques valeurs initiales.

Aujourd'hui, nous allons personnaliser le module `job` que nous avons créé hier. Actuellement, ce module a déjà tout le code utile pour Jobeet :

- Une page listant tous les jobs
- Une page pour créer un nouveau job
- Une page pour mettre à jour une job déjà existant
- Une page pour supprimer un job

Bien que le code est prêt à être utilisé tel qu'il est, nous devons modifier les Templates afin que nos pages correspondent à notre maquette.

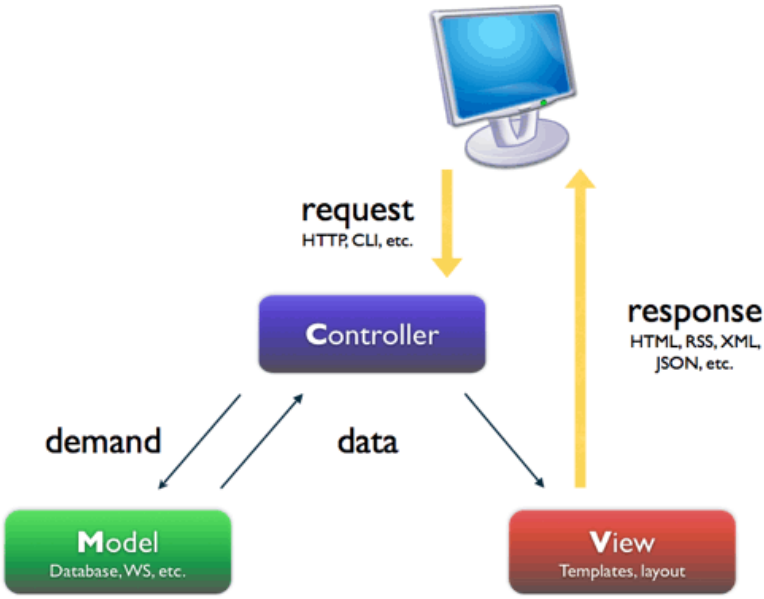
L'architecture MVC

Si vous avez l'habitude de développer des sites web en PHP sans framework, vous avez probablement utilisé le principe d'un fichier PHP par page HTML. Ces fichiers PHP ayant très certainement une structure proche de : l'initialisation et la configuration globale, le traitement associé à la page demandée, la récupération des données depuis la base de données et enfin la mise en place du code HTML formant la page.

Vous pouvez également utiliser un moteur de Template permettant de séparer la logique du HTML. Vous utilisez peut-être une couche d'abstraction permettant de séparer l'interaction du modèle avec celui du traitement des données. Mais la plupart du temps, vous vous retrouvez avec beaucoup de code absolument cauchemardesque à maintenir. Le code a rapidement été mis en place, mais la plupart du temps, ce dernier est de plus en plus difficile à modifier, notamment parce que personne, excepté vous, ne sait comment votre site a été conçu et comment il fonctionne.

Comme toujours : à chaque problème, ses solutions. Pour le développement Web, les solutions les plus populaires pour organiser votre code de nos jours est la mise en place d'une **architecture MVC**. En résumé, l'architecture MVC définit un cadre d'organisation de votre code en accord avec sa nature. Ce modèle permet une séparation de votre code en **trois couches** :

- La couche **Modèle** contenant le traitement logique de vos données (les accès à la base de données se trouvent dans cette couche). Vous savez déjà que symfony stocke toutes les classes et tous les fichiers relatifs au Modèle dans le répertoire `lib/model`.
- La **Vue** est la couche où interagit l'utilisateur (un moteur de template fait parti de cette couche). Dans symfony, la couche vue est principalement faite de Templates PHP. Ces fichiers sont stockés dans les différents dossiers `templates/` comme nous le verrons plus loin.
- La **Contrôleur** est un morceau de code qui appelle le modèle pour obtenir certaines données qu'il passe à la Vue pour le rendu au client. Quand nous avons installé symfony le premier jour, nous avons vu que toutes les requêtes étaient gérées par des contrôleurs frontaux (`index.php` et `frontend_dev.php`). Ces contrôleurs frontaux délèguent le réel travail à des **actions**. Comme nous l'avons vu hier, ces actions sont logiquement regroupées dans des **modules**.



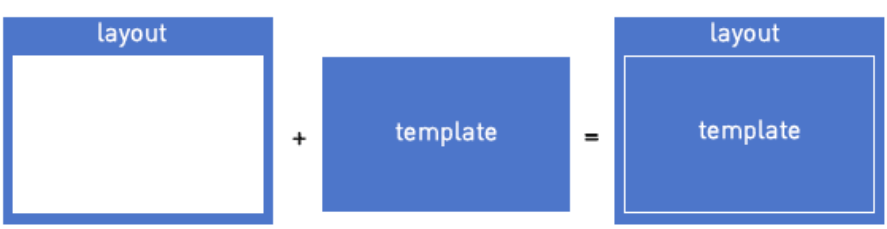
Aujourd'hui, nous allons utiliser la maquette définie le 2ième jour afin de personnaliser la page d'accueil et la page d'un emploi. Nous allons les rendre dynamique. En chemin, nous allons ajuster un tas de choses dans beaucoup de fichiers différents afin de montrer la structure de répertoire de symfony et la manière de séparer le code entre les couches.

La mise en page

D'abord, si vous regardez de plus près la maquette, vous remarquerez que la quantité de chaque page vous semble le même. Vous savez déjà que la duplication de code est mauvais, si nous parlons de code HTML ou PHP, donc nous devons trouver un moyen d'empêcher ces éléments communs de la vue d'aboutir à la duplication du code.

Une manière de résoudre le problème est de définir une entête et un pied de page et de les inclure dans chaque Template :

Mais dans ce cas, les fichiers header et footer ne contiennent pas de code HTML valide. Il doit y avoir un meilleur moyen. Plutôt que de réinventer la roue, nous allons utiliser un autre modèle pour résoudre ce problème : le modèle décorateur. Le modèle décorateur résout le problème d'une manière différente : le Template est décorée après que le contenu soit mise en page grâce à un template global, appelé **layout** dans symfony :



La mise en page par défaut d'une application est appelée `layout.php` et se trouve dans le dossier `apps/frontend/templates/`. Ce répertoire contient tous les Templates globaux pour une application.

Remplacez le contenu par défaut du layout par le code suivant :

```
<!-- apps/frontend/templates/layout.php -->
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Jobeet - Your best job board</title>
    <link rel="shortcut icon" href="/favicon.ico" />
    <?php include_javascripts() ?>
    <?php include_stylesheets() ?>
  </head>
  <body>
    <div id="container">
      <div id="header">
        <div class="content">
          <h1><a href="<?php echo url_for('job/index') ?>">
            <img src="/images/logo.jpg" alt="Jobeet Job Board" />
          </a></h1>

          <div id="sub_header">
            <div class="post">
              <h2>Ask for people</h2>
              <div>
                <a href="<?php echo url_for('job/index') ?>">Post a Job</a>
              </div>
            </div>

            <div class="search">
              <h2>Ask for a job</h2>
              <form action="" method="get">
                <input type="text" name="keywords"
                  id="search_keywords" />
                <input type="submit" value="search" />
                <div class="help">
                  Enter some keywords (city, country, position, ...)
                </div>
              </form>
            </div>
          </div>
        </div>
      </div>

      <div id="content">
        <?php if ($sf_user->hasFlash('notice')): ?>
          <div class="flash_notice">
            <?php echo $sf_user->getFlash('notice') ?>
          </div>
        <?php endif; ?>

        <?php if ($sf_user->hasFlash('error')): ?>
          <div class="flash_error">
            <?php echo $sf_user->getFlash('error') ?>
          </div>
        <?php endif; ?>

        <div class="content">
          <?php echo $sf_content ?>
        </div>
      </div>

      <div id="footer">
        <div class="content">
          <span class="symfony">
            <img src="/images/jobeet-mini.png" />
            powered by <a href="http://www.symfony-project.org/">
            <img src="/images/symfony.gif" alt="symfony framework" />
            </a>
          </span>
          <ul>
            <li><a href="">About Jobeet</a></li>
            <li class="feed"><a href="">Full feed</a></li>
            <li><a href="">Jobeet API</a></li>
            <li class="last"><a href="">Affiliates</a></li>
          </ul>
```

```
        </div>
      </div>
    </div>
  </body>
</html>
```

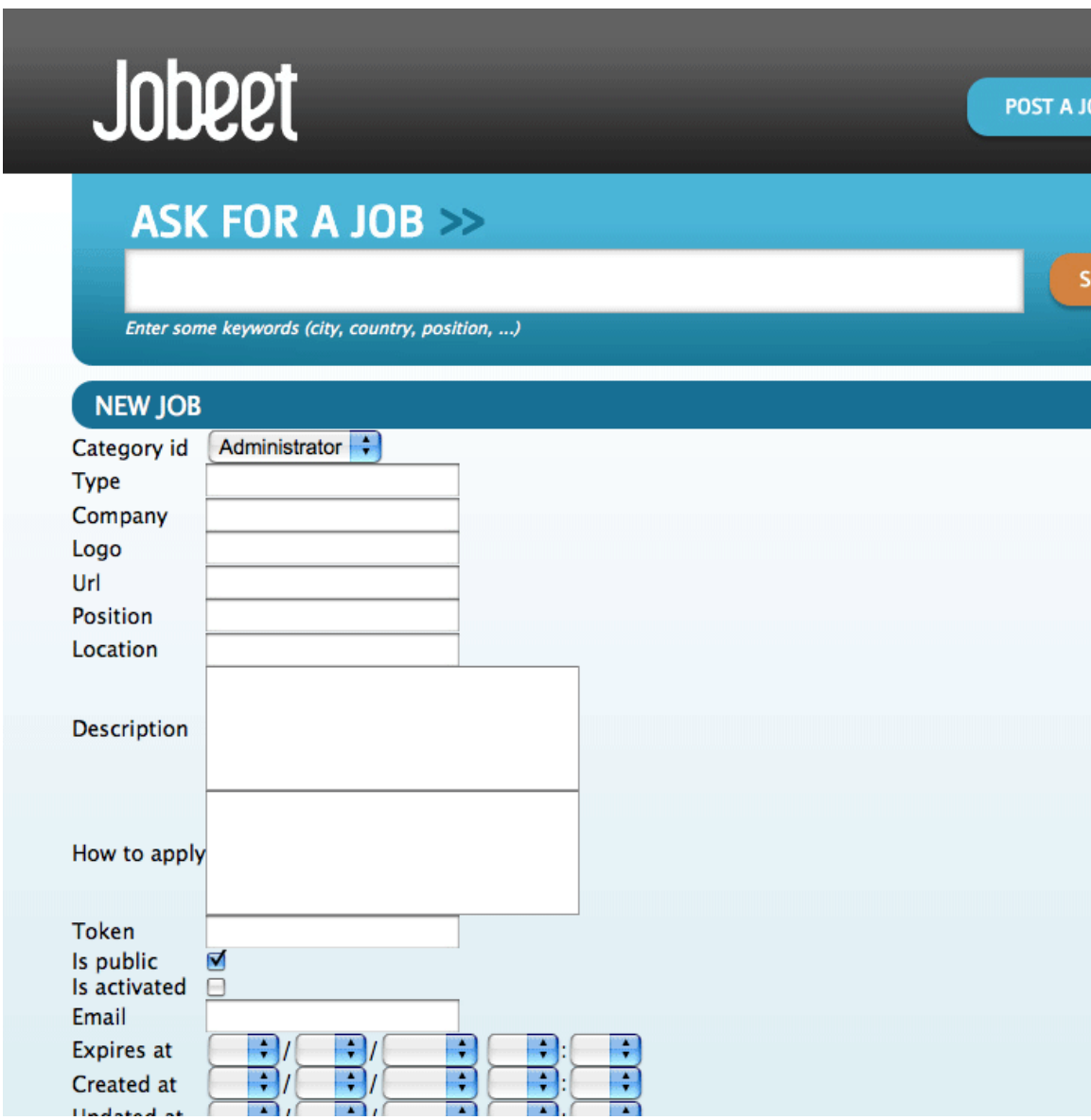
Un template symfony est juste un fichier PHP. Dans le template layout, vous trouverez des appels à des fonctions PHP et des références à des variables PHP. `$sf_content` est la variable la plus intéressante : elle est définie par le framework lui-même et contient le code HTML généré par l'action.

Si vous parcourez le module `job` (`http://www.jobeet.com.localhost/frontend_dev.php/job`), vous verrez que toutes les actions sont décorés par le layout.

Les feuilles de style, les images et les Javascripts

Comme ce tutoriel n'est pas sur le design web, nous avons déjà préparé toutes les ressources que nous utiliserons pour Jobeet : téléchargez les fichiers image et mettez les dans le dossier `web/images/`; téléchargez les fichiers feuilles de style et mettez les dans le dossier `web/css/`.

Dans le layout, nous avons inclus un *favicon*. Vous pouvez télécharger celle de Jobeet et la déposer dans le dossier `web/`.



Par défaut, la tâche `generate:project` a créé trois dossiers pour les ressources du projet :

web/images/ pour les images, web/~css|CSS~/ pour les feuilles de style et web/js/ pour les JavaScripts. Ceci fait partie des nombreuses conventions définies par symfony, mais vous pouvez évidemment les placer dans un autre dossier sous le répertoire web/.

Un lecteur avisé aura remarqué que, bien que le fichier main.css n'est défini nul part dans le layout par défaut, il est nécessairement présent dans le code HTML généré. Mais pas pour les autres. Comment est-ce possible ?

La feuille de style a été incluse grâce à la fonction include_stylesheets() située entre les balises <head> du fichier layout. La fonction include_stylesheets() est appelée un **helper**. Un helper est une fonction, définie par symfony, pouvant prendre des paramètres et renvoyant du code HTML. La plupart du temps, les helpers permettent de gagner du temps, ils contiennent du code fréquemment utilisé dans les templates. Le helper include_stylesheets() génère une balise <link> spécifique aux feuilles de style.

Mais comment le helper sait quelle feuille de style inclure ?

La couche de la Vue peut être paramétrée en éditant le fichier de configuration view.yml de l'application. Voici le fichier par défaut généré lors de l'appel par la tâche generate:app :

```
# apps/frontend/config/view.yml
default:
  http_metas:
    content-type: text/html

  metas:
    #title:        symfony project
    #description:  symfony project
    #keywords:     symfony, project
    #language:     en
    #robots:       index, follow

  stylesheets:    [main.css]

  javascripts:    []

  has_layout:     true
  layout:         layout
```

Le fichier view.yml configure les paramètres par défaut pour tous les Templates de l'application. Par exemple, l'entrée stylesheets définit un tableau de fichiers de feuille de style à inclure pour chaque page de l'application (ceci grâce au helper include_stylesheets()).

Dans le fichier de configuration par défaut view.yml, le fichier référencé est main.css, et non pas css/main.css. En fait, les deux définitions sont équivalentes. Symfony préfixe les chemins relatifs avec /css/.

Si plusieurs fichiers sont définis, symfony les inclura dans le même ordre que celui dans lequel ils ont été définis :

```
stylesheets:    [main.css, jobs.css, job.css]
```

Vous pouvez également définir l'attribut media et omettre le suffixe .css :

```
stylesheets:    [main.css, jobs.css, job.css, print: { media: print }]
```

Cette configuration génèrera le code suivant :

```
<link rel="stylesheet" type="text/css" media="screen"
  href="/css/main.css" />
<link rel="stylesheet" type="text/css" media="screen"
  href="/css/jobs.css" />
<link rel="stylesheet" type="text/css" media="screen"
  href="/css/job.css" />
<link rel="stylesheet" type="text/css" media="print"
  href="/css/print.css" />
```

Le fichier de configuration view.yml définit également le layout utilisé par défaut pour l'application. Par défaut, son nom est layout. Par conséquent, symfony met en page chacune de vos pages à partir du fichier layout.php. Vous pouvez également désactiver cette mise en

page en définissant l'entrée has_layout à false.

Il fonctionne comme jobs.css, mais le fichier n'est nécessaire que pour la page d'accueil et le fichier job.css n'est nécessaire que pour la page emploi. Le fichier de configuration view.yml peut être personnalisés sur la base de chaque module. Changez la clé stylesheets du fichier view.yml de l'application pour contenir uniquement le fichier main.css :

```
# apps/frontend/config/view.yml
stylesheets:    [main.css]
```

Pour personnaliser la vue du module job, créez un fichier view.yml dans le répertoire apps/frontend/modules/job/config/ :

```
# apps/frontend/modules/job/config/view.yml
indexSuccess:
  stylesheets: [jobs.css]

showSuccess:
  stylesheets: [job.css]
```

Sous les sections indexSuccess et showSuccess (qui sont les noms des Templates associés aux actions index et show, comme nous le verrons plus tard), vous pouvez personnaliser les entrées se trouvant sous la section default du fichier view.yml de l'application. Toutes les entrées spécifiques sont fusionnées avec la configuration de l'application. Vous pouvez également définir une configuration de toutes les actions d'un module avec la section spéciale all.

Principes de configuration dans symfony

Pour beaucoup de fichiers de configuration de symfony, un même paramètre peut être défini à différents niveaux :

- La configuration par défaut se trouve dans le framework
- La configuration globale pour le projet (dans le répertoire config/)
- La configuration locale pour l'application (dans le répertoire apps/APP/config/)
- La configuration locale est restreinte à un module (dans le répertoire apps/APP/modules/MODULE/config/)

Lors de l'exécution, le système de configuration fusionne tous les valeurs depuis les différents fichiers si ils existent et met le résultat en cache pour de meilleures performances.

En règle générale, quand quelque chose est configurable via un fichier de configuration, la même chose peut être faite avec du code PHP. Au lieu de créer un fichier view.yml pour le module job par exemple, vous pouvez aussi utiliser le helper use_stylesheet() pour inclure une feuille de style depuis un template :

```
<?php use_stylesheet('main.css') ?>
```

Vous pouvez également utiliser ce helper dans le layout pour inclure une feuille de style globale.

Le choix entre une méthode ou une autre est réellement une question de goût. Le fichier view.yml permet de définir quelque chose pour toutes les actions d'un module, ce qui n'est pas possible depuis un template. Cela dit, la configuration est plus statique. A l'inverse, le helper use_stylesheet() est plus flexible et plus encore, tout se trouve au même endroit : la définition des feuilles de style et le code HTML. Pour Jobeet, nous allons utiliser le helper use_stylesheet(), nous pouvons donc supprimer le fichier view.yml que nous venons de créer, et mettre à jour le template job avec les appels à use_stylesheet() :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<?php use_stylesheet('jobs.css') ?>

<!-- apps/frontend/modules/job/templates/showSuccess.php -->
<?php use_stylesheet('job.css') ?>
```

De la même manière, la configuration JavaScript est faite via l'entrée javascripts du fichier de configuration view.yml et via le helper use_javascript() permettant d'inclure des fichiers JavaScript dans un Template.

La page d'accueil Job

Comme vu le troisième jour, la page d'accueil est générée par l'action `index` du module `job`. L'action `index` fait partie de la couche Contrôleur de la page et le template associé, `indexSuccess.php`, fait parti de la couche Vue :

```
apps/
  frontend/
    modules/
      job/
        actions/
          actions.class.php
        templates/
          indexSuccess.php
```

L'action

Chaque action est représentée par une méthode de la classe. Pour la page d'accueil job, la classe est `jobActions` (le nom du module avec le suffixe `Actions`) et la méthode est `executeIndex()` (le nom de l'action avec le préfixe `execute`). Dans notre cas, cela renvoie tous les jobs de la base de données :

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeIndex(sfWebRequest $request)
  {
    $this->jobeet_jobs = Doctrine::getTable('JobeetJob')
      ->createQuery('a')
      ->execute();
  }

  // ...
}
```

Analysons de plus près le code : la méthode `executeIndex()` (le Contrôleur) appelle la Table `JobeetJob` pour créer une requête renvoyant tous les jobs. La Table renvoie une 'Doctrine_Collectiond'objets de typeJobettJobque l'on affecte à la propriétéjobeet_jobs` de l'objet courant.

Toutes les propriétés des objets sont automatiquement passées au template (la Vue). Pour transmettre des données du Contrôleur à la Vue, il vous suffit simplement de créer une nouvelle propriété :

```
public function executeFooBar(sfWebRequest $request)
{
  $this->foo = 'bar';
  $this->bar = array('bar', 'baz');
}
```

Cette méthode rendra les variables `$foo` et `$bar` accessibles depuis le template.

Le Template

Par défaut, le nom du template associé à l'action est déduit par symfony grâce a une convention (le nom de l'action avec le suffixe `Success`).

Le template `indexSuccess.php` génère une table HTML pour tous les jobs. Voici le code du Template actuel :

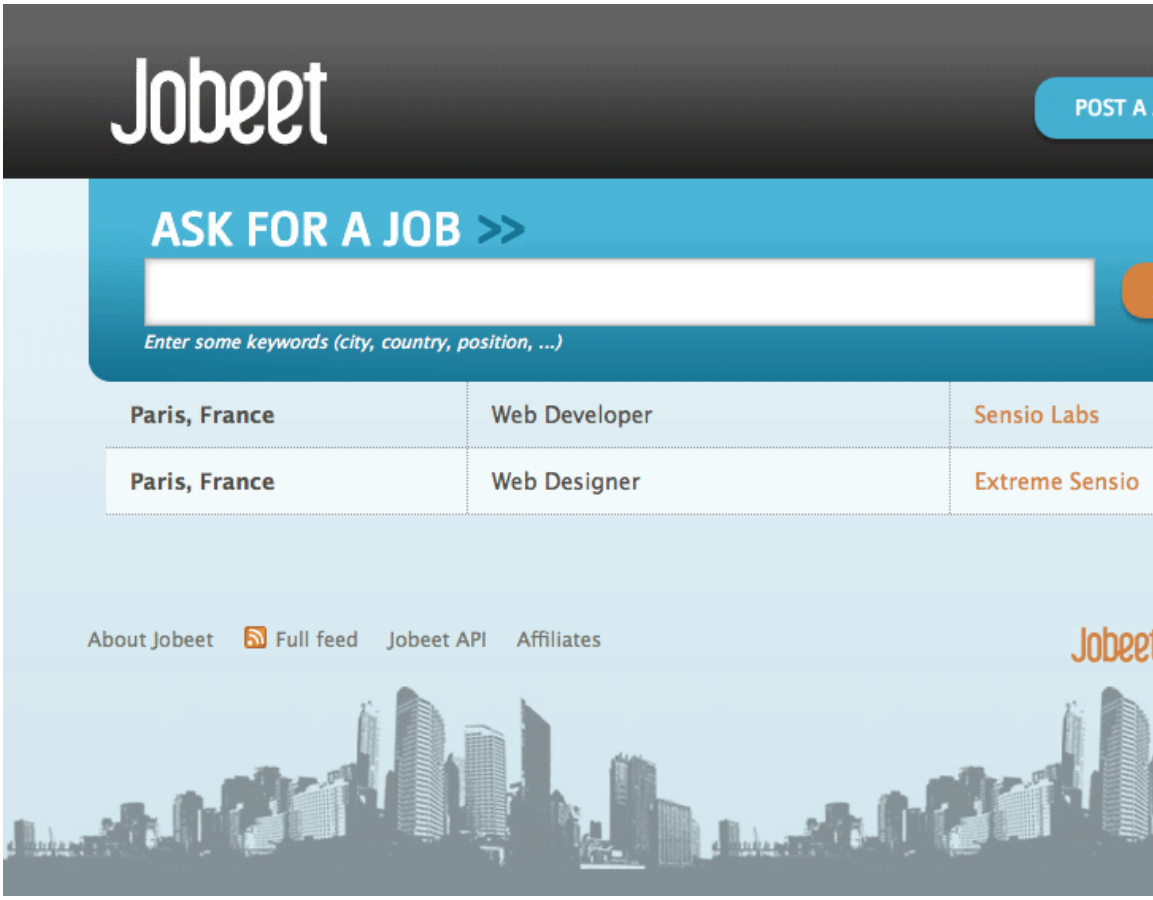
```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<?php use_stylesheet('jobs.css') ?>

<h1>Job List</h1>

<table>
  <thead>
    <tr>
      <th>Id</th>
      <th>Category</th>
      <th>Type</th>
<!-- more columns here -->
      <th>Created at</th>
      <th>Updated at</th>
```

```
    </tr>
  </thead>
  <tbody>
    <?php foreach ($jobeet_jobs as $jobeet_job): ?>
    <tr>
      <td>
        <a href="<?php echo url_for('job/show?id='.$jobeet_job->getId()) ?>">
          <?php echo $jobeet_job->getId() ?>
        </a>
      </td>
      <td><?php echo $jobeet_job->getCategoryId() ?></td>
      <td><?php echo $jobeet_job->getType() ?></td>
<!-- more columns here -->
      <td><?php echo $jobeet_job->getCreatedAt() ?></td>
      <td><?php echo $jobeet_job->getUpdatedAt() ?></td>
    </tr>
    <?php endforeach; ?>
  </tbody>
</table>

<a href="<?php echo url_for('job/new') ?>">New</a>
```

Dans ce code, la boucle `foreach` parcourt la liste d'objets `job` (`$jobeet_jobs`) et pour chaque job, chaque valeur de colonne est affichée. Souvenez-vous, pour accéder à la valeur d'une colonne, il suffit simplement d'appeller une méthode accesseur. dont le nom commence par `get` et suivit du nom de la colonne en camelCased (par exemple, la méthode `getCreatedAt()` permet d'accéder à la colonne `created_at`).

Faisons un peu de tri dans tout ça afin de n'afficher qu'une partie des colonnes :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<?php use_stylesheet('jobs.css') ?>

<div id="jobs">
  <table class="jobs">
    <?php foreach ($jobeet_jobs as $i => $job): ?>
      <tr class="<?php echo fmod($i, 2) ? 'even' : 'odd' ?>">
        <td class="location"><?php echo $job->getLocation() ?></td>
        <td class="position">
          <a href="<?php echo url_for('job/show?id='.$job->getId()) ?>">
            <?php echo $job->getPosition() ?>
          </a>
        </td>
        <td class="company"><?php echo $job->getCompany() ?></td>
      </tr>
    <?php endforeach; ?>
  </table>
</div>
```



La fonction `url_for()` utilisée dans ce template est un helper symfony que nous détaillerons dans le chapitre de demain.

Le template de la page job

Personnalisons maintenant le template de la page job. Ouvrez le fichier showSuccess.php et remplacez son contenu par le code suivant :

```
<!-- apps/frontend/modules/job/templates/showSuccess.php -->
<?php use_stylesheet('job.css') ?>
<?php use_helper('Text') ?>

<div id="job">
  <h1><?php echo $job->getCompany() ?></h1>
  <h2><?php echo $job->getLocation() ?></h2>
  <h3>
    <?php echo $job->getPosition() ?>
    <small> - <?php echo $job->getType() ?></small>
  </h3>

  <?php if ($job->getLogo()): ?>
    <div class="logo">
      <a href="<?php echo $job->getUrl() ?>">
        <img src="/uploads/jobs/<?php echo $job->getLogo() ?>"
          alt="<?php echo $job->getCompany() ?> logo" />
      </a>
    </div>
  <?php endif; ?>

  <div class="description">
    <?php echo simple_format_text($job->getDescription()) ?>
  </div>

  <h4>How to apply?</h4>

  <p class="how_to_apply"><?php echo $job->getHowToApply() ?></p>

  <div class="meta">
```

```
    <small>posted on <?php echo $job->getDateTimeObject('created_at')->format('m/d/Y')
?></small>
  </div>

  <div style="padding: 20px 0">
    <a href="<?php echo url_for('job/edit?id='.$job->getId()) ?>">
      Edit
    </a>
  </div>
</div>
```

Ce template utilise la variable `$job` passée en paramètre par l'action pour afficher les informations sur un job. Comme nous avons renommé la variable utilisée dans le template de `$jobeet_job` en `$job`), vous devez également faire modification dans l'action show (attention, la variable s'y trouve deux fois) :

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeShow(sfWebRequest $request)
{
  $this->job = Doctrine::getTable('JobeetJob')-> find($request->getParameter('id'));
  $this->forward404Unless($this->job);
}
```

Notez que les colonnes de date peuvent être converties en instances d'objets PHP DateTime. Comme nous avons défini la colonne `created_at` comme un timestamp, vous pouvez convertir l valeur de la colonne en un objet DateTime en utilisant la méthode `getDateTimeObject()` et ensuite appeler la méthode `format()` dont son premier argument prend un format de mise en forme d'une date :

```
$job->getDateTimeObject('created_at')->format('m/d/Y');
```

La description d'un job utilise le helper `simple_format_text()` afin de formater le texte en HTML, en remplaçant notamment les retours chariots par des balises `<br />`. Comme ce helper fait parti du groupe Text et que celui-ci n'est pas chargé par défaut, nous le chargeons manuellement en utilisant le helper `use_helper()`.

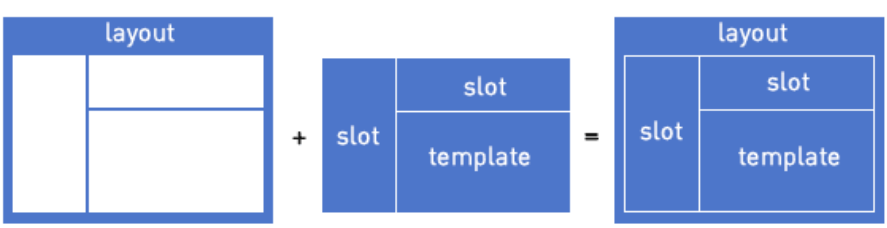
Les Slots

Actuellement, le titre de toutes les pages est défini dans la balise `<title>` du layout :

```
<title>Jobeet - Your best job board</title>
```

Mais pour un job, nous aimerions avoir des informations plus utiles telles que le nom de la société et le type d'emploi.

Avec symfony, quand une zone du layout dépend du template à afficher, vous devez utiliser un slot :



Ajoutez un slot au layout afin de rendre le titre dynamique :

```
// apps/frontend/templates/layout.php
<title><?php include_slot('title') ?></title>
```

Chaque slot est identifié par un nom (ici `title`) et peut être affiché en utilisant le helper `include_slot()`. Maintenant, au début du template showSuccess.php, utilisez le helper `slot()` afin de définir le contenu du slot pour la page job :

```
// apps/frontend/modules/job/templates/showSuccess.php
<?php slot(
  'title',
  sprintf('%s is looking for a %s', $job->getCompany(), $job->getPosition()))
?>
```

Si le titre est complexe à définir, le helper `slot()` peut aussi être utilisé dans un block de code :

```
// apps/frontend/modules/job/templates/showSuccess.php
<?php slot('title') ?>
  <?php echo sprintf('%s is looking for a %s', $job->getCompany(), $job-
>getPosition()) ?>
<?php end_slot(); ?>
```

Pour certaines pages, comme la page d'accueil, nous avons juste besoin d'un titre générique. Plutôt que de répéter le même titre encore et encore dans chaque template, nous pouvons définir un titre par défaut dans le layout :

```
// apps/frontend/templates/layout.php
<title>
  <?php include_slot('title', 'Jobeet - Your best job board') ?>
</title>
```

Le deuxième argument de la méthode `include_slot()` est la valeur par défaut pour le slot si elle n'a pas été défini. Si la valeur par défaut est plus longue ou a certaines balises HTML, vous pouvez aussi le définir comme dans le code suivant :

```
// apps/frontend/templates/layout.php
<title>
  <?php if (!include_slot('title')): ?>
    Jobeet - Your best job board
  <?php endif; ?>
</title>
```

Le helper `include_slot()` renvoie `true` si le slot a été défini. Ainsi, lorsque vous spécifiez une valeur pour le slot `title` dans un template, il est utilisé, sinon, ce sera le titre par défaut qui sera utilisé.

Nous avons déjà vu quelques helpers commençant par `include_`. Ces helpers renvoient du code HTML et dans la plupart des cas, ils ont un helper `get_` permettant de renvoyer uniquement le contenu :

```
<?php include_slot('title') ?>
<?php echo get_slot('title') ?>

<?php include_stylesheets() ?>
<?php echo get_stylesheets() ?>
```

L'action de la page job

La page job est générée grâce à l'action show, définie par la méthode `executeShow()` du module `job` :

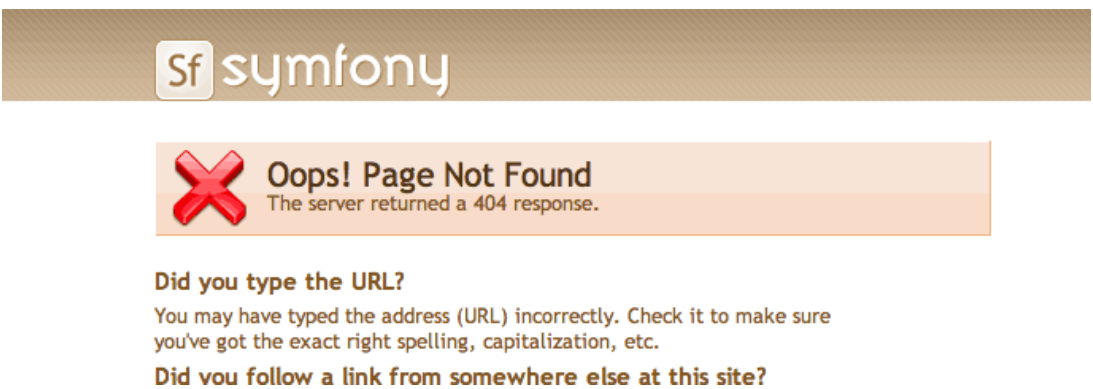
```
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = Doctrine::getTable('JobeetJob')-> find($request->getParameter('id'));
    $this->forward404Unless($this->job);
  }

  // ...
}
```

Comme dans l'action `index`, la classe de la table `JobeetJob` est utilisée pour récupérer un job, cette fois en utilisant la méthode `find()`. Le paramètre de cette méthode est l'identifiant unique d'un job, sa clé primaire. La prochaine section explique pourquoi l'instruction `$request->getParameter('id')` renvoie la clé primaire d'un job.

Si le job n'existe pas dans la base de données, nous voudrions renvoyer l'utilisateur vers une page 404, c'est ce que fait exactementla méthode `forward404Unless()`. Elle prend en premier

paramètre un Booléen et, à moins que ce paramètre ne soit à true, elle arrête l'exécution normale. Cette méthode génère une exception `sfError404Exception` et vous n'avez donc pas besoin de rajouter de `return` après cette méthode.

Comme pour toutes exceptions, la page affichée est différente en fonction de l'environnement de prod ou de dev :



Avant que nous déployons notre site Jobeet sur le serveur de production, vous apprendrez à personnaliser la page 404 par défaut.

La famille des méthodes "forward"

L'appel à la méthode `forward404Unless` est équivalent à :

```
$this->forward404If(!$this->job);
```

qui est équivalent à :

```
if (!$this->job)
{
  $this->forward404();
}
```

La méthode `forward404()` elle-même étant juste un raccourci pour :

```
$this->forward('default', '404');
```

La méthode `forward()` renvoie vers une autre action de la même application; dans l'exemple précédent, vers l'action 404 du module `default`. Le module `default` fait parti intégrante de symfony et fournit des actions par défaut pour afficher les pages 404, de sécurité et de connexion.

Les requêtes et les réponses

Quand vous accédez à la page `/job` ou `/job/show/id/1` depuis votre navigateur, vous provoquez un ensemble de traitements entre le serveur web et votre ordinateur. Votre navigateur envoie une **requête** et le serveur vous renvoie une **réponse**.

Nous avons déjà vu que symfony encapsule les requêtes dans un objet `sfWebRequest` (regardez la signature de la méthode `executeShow()`). Et comme symfony est un framework Orienté

Objet, la réponse est également un objet de classe `sfWebResponse`. Vous pouvez récupérer l'objet de la réponse dans une action en appelant `$this->getResponse()`.

Ces objets permettent un accès pratique et simple pour obtenir des informations sur des fonctions PHP et des variables globales PHP.

Pourquoi symfony redéfinit-il des fonctions PHP déjà existantes ? Premièrement, parce que celles de symfony sont plus puissantes que leur homologue PHP. Ensuite, parce que quand vous testez une application, il est plus facile de simuler des requêtes ou des réponses grâce à des objets plutôt que d'essayer d'utiliser des variables globales ou travailler avec des fonctions PHP comme `header()` si mystiques.

La requête

La classe `sfWebRequest` redéfinit les tableaux globaux PHP `$_SERVER`, `$_COOKIE`, `$_GET`, `$_POST`, et `$_FILES` :

Nom de la méthode	Équivalent PHP
`getMethod()`	`$_SERVER['REQUEST_METHOD']`
`getUri()`	`$_SERVER['REQUEST_URI']`
`getReferer()`	`$_SERVER['HTTP_REFERER']`
`getHost()`	`$_SERVER['HTTP_HOST']`
`getLanguages()`	`$_SERVER['HTTP_ACCEPT_LANGUAGE']`
`getCharsets()`	`$_SERVER['HTTP_ACCEPT_CHARSET']`
`isXmlHttpRequest()`	`$_SERVER['X_REQUESTED_WITH'] == 'XMLHttpRequest'`
`getHttpHeader()`	`$_SERVER`
`getCookie()`	`$_COOKIE`
`isSecure()`	`$_SERVER['HTTPS']`
`getFiles()`	`$_FILES`
`getGetParameter()`	`$_GET`
`getPostParameter()`	`$_POST`
`getUrlParameter()`	`$_SERVER['PATH_INFO']`
`getRemoteAddress()`	`$_SERVER['REMOTE_ADDR']`

Nous avons déjà accédé aux paramètres d'une requête en utilisant la méthode `getParameter()`. Elle renvoie une valeur depuis la variable globale `$_GET` ou `$_POST`, ou depuis la variable `PATH_INFO`.

Si vous voulez être sûr qu'un paramètre demandé provienne de l'une de ces variables en particulier, vous devez utiliser respectivement la méthode `getGetParameter()`, `getPostParameter()` et `getUrlParameter()`.

Quand vous voulez restreindre une action pour une méthode HTTP spécifique, par exemple quand vous voulez être sûr qu'un formulaire ait été envoyé via la méthode `POST`, vous pouvez utiliser la méthode `isMethod()` : `$this->forwardUnless($request->isMethod('POST'));`.

La réponse

La classe `sfWebResponse` redéfinit les méthodes PHP `~header|HTTP Headers~()` et `setraw~cookie|Cookies~()` :

Nom de la méthode	Équivalent PHP
`setCookie()`	`setrawcookie()`
`setStatusCode()`	`header()`
`setHttpHeader()`	`header()`
`setContentType()`	`header()`

`addVaryHttpHeader()`	`header()`
`addCacheControlHttpHeader()`	`header()`

Évidemment, la classe `sfWebResponse` permet aussi de définir la réponse du serveur web (`setContent()`) et de l'envoyer au navigateur (`send()`).

Plus tôt aujourd'hui, nous avons vu comment gérer les feuilles de style et les JavaScripts dans le fichier `view.yml` et dans les templates. Finalement, ces deux techniques utilisent les méthodes `addStylesheet()` et `addJavascript()` de l'objet réponse.

Les classes `sfAction`, `sfRequest`, et `sfResponse` fournissent un grand nombre de méthodes très utiles. N'hésitez pas à parcourir la documentation de l'API pour en apprendre plus sur les classes internes de symfony.

Conclusion

Tout au long de ce chapitre, nous avons décrit quelques uns des modèles de conception utilisés par symfony. Espérons que la structure des répertoires du projet ait maintenant plus de sens. Nous avons joué avec les Templates en manipulant la mise en page et les fichiers des Templates. Nous avons également rendu les pages un peu plus dynamiques grâce aux slots et aux actions.

Au cours du prochain chapitre, nous en apprendrons davantage sur le helper `url_for()` que nous avons aperçu aujourd'hui, et le "sous-framework" de routage qui lui est associé.

« Jour 3 : Le modèle de données

Jour 5 : Le Routage »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 5 : Le Routage

Si vous avez terminé le 4ème jour, vous connaissez maintenant le modèle MVC, cela doit vous paraître de plus en plus comme la meilleure manière de coder. Prenez le temps de le comprendre et vous ne le regretterez pas. Pour pratiquer un peu hier, nous avons personnalisé les pages Jobeet et dans le processus, nous avons également examiné plusieurs concepts de symfony, comme la mise en page, les helpers et les slots

Aujourd'hui nous explorerons le monde merveilleux du routage avec symfony.

Les URLs

Si vous cliquez sur un emploi sur la page d'accueil Jobeet, l'URL ressemble à ceci : `/job/show/id/1`. Si vous avez déjà développé des sites Web PHP, vous êtes probablement plus habitués à des URL comme `/job.php?id=1`. Comment symfony fait pour que cela fonctionne ? Comment symfony détermine quelle est l'action à appeler pour cette URL ? Pourquoi on récupère l'`id` du job avec `$request->getParameter('id')` ? Aujourd'hui, nous allons répondre à toutes ces questions.

Tout d'abord, voyons ce que sont exactement les URLs. Dans un contexte web, une URL est l'identifiant unique d'un contenu web. Quand vous allez sur une URL, vous demandez au navigateur d'aller chercher un contenu identifié par cette URL. Ainsi, comme l'URL est l'interface entre le site et l'utilisateur, elle doit donner des informations utiles sur le contenu qu'elle référence. Toutefois, les URLs "traditionnelles" ne décrivent pas vraiment le contenu, elles exposent la structure interne de l'application. L'utilisateur ne se soucie pas que votre site est développé avec le langage PHP, ou qu'un emploi a un certain identifiant dans la base de données. Exposer le fonctionnement interne de votre application est aussi très mauvais en ce qui concerne la sécurité : que faire si l'utilisateur essaie de deviner l'adresse URL pour le contenu auquel il n'a pas accès ? Bien sûr, le développeur doit sécuriser ces accès, mais la meilleure solution est de cacher les informations sensibles.

Les URLs sont si importantes, que symfony possède un framework dédié à leur gestion : le framework de **routage**. Le routage gère aussi bien les URIs internes que les URLs externes. Après une requête, le routage analyse l'URL et la convertit en URI interne.

Vous avez déjà vu une URI interne sur la page job dans le Template `indexSuccess.php` :

```
'job/show?id='.$job->getId()
```

Le helper `url_for()` convertit cette URI interne en une URL propre :

```
/job/show/id/1
```

L'URI interne se compose de plusieures parties : `job` qui est le module, `show` qui est l'action et les paramètres servants à la requête passée à l'action. Le modèle générique pour une URI interne est :

```
MODULE/ACTION?key=value&key_1=value_1&...
```

Comme le routage de symfony est un processus bi-directionnel. Vous pouvez changer les URLs sans changer l'exécution technique. C'est l'un des avantages principaux du front-controller design pattern.

Configuration du Routage

Le lien entre les URIs internes et les URLs externes se paramètre dans le fichier de configuration `routing.yml` :

```
# apps/frontend/config/routing.yml
homepage:
```

```
  url:   /
  param: { module: default, action: index }

default_index:
  url:   /:module
  param: { action: index }

default:
  url:   /:module/:action/*
```

Le fichier `routing.yml` décrit les routes. Une route possède un nom (`homepage`), un modèle (`/:module/:action/*`) et d'autres paramètres (après la clé `param`).

A l'appel d'une requête, le routage essaye de faire correspondre un modèle avec l'URL donnée. La première route trouvée est utilisée, c'est pourquoi l'ordre dans `routing.yml` est important. Voyons quelques exemples pour mieux comprendre comment le routage fonctionne.

A l'appel de la page d'accueil de Jobeet, avec l'URL `/job`, la première route qui correpond est `default_index`. Dans le modèle, un mot qui est préfixé de deux points (`:`) est une variable. Donc le modèle `/:module` signifie : trouver un `/` suivi par quelquechose. Dans notre exemple, la variable `module` aura la valeur `job`. Cette valeur peut alors être recherchée avec `$request->getParameter('module')` dans l'action. Cette route définit aussi une valeur par défaut pour la variable `action`. Donc toutes les URLs correspondantes à cette route possèderont par défaut le paramètre `action` avec la valeur `index`.

Si vous allez sur la page `/job/show/id/1`, symfony fera correspondre le dernier modèle : `/:module/:action/*`. Dans un modèle, une étoile (`*`) correspond à une collection de paires de variable/valeur séparées par des slashes (`/`) :

Paramètre de la requête	Valeur
module	job
action	show
id	1

Les variables `module` et `action` sont spéciales car elles sont utilisées par symfony pour déterminer l'action à exécuter.

L'URL `/job/show/id/1` peut être créee dans un Template en utilisant le helper `url_for()` :

```
url_for('job/show?id='.$job->getId())
```

Vous pouvez aussi utiliser le nom de la route en le faisant précéder par `@` :

```
url_for('@default?module=job&action=show&id='.$job->getId())
```

Les deux appels sont équivalents, mais celui-ci est beaucoup plus rapide car le routage n'a pas besoind'analyser toutes les routes pour trouver la meilleure correspondance, et elle est moins attachée à l'exécution (les noms du module et de l'action ne sont pas présents dans l'URI interne).

Personnalisation des Routes

Pour l'instant, lorsque vous demandez l'URL `/` dans un navigateur, vous avez la page par défaut de félicitation de symfony. C'est parce que cette URL correspond à la route `homepage`. Mais il est logique de le changer la page d'accueil Jobeet. Pour faire le changement, modifiez la variable `module` de la route `homepage` par `job` :

```
# apps/frontend/config/routing.yml
homepage:
  url:   /
  param: { module: job, action: index }
```

Nous pouvons maintenant modifier le lien du logo Jobeet dans le layout en utilisant la route `homepage` :

```
<!-- apps/frontend/templates/layout.php -->
<h1>
  <a href="<?php echo url_for('@homepage') ?>">
    <img src="/images/logo.jpg" alt="Jobeet Job Board" />
  </a>
```

```
</h1>
```

C'était facile !

Lorsque vous mettez à jour la configuration du routage, les modifications sont immédiatement prises en compte dans l'environnement de développement. Mais pour les faire fonctionner aussi dans l'environnement de production, vous devez vider le cache en appelant la tâche `cache:clear`.

Pour avoir quelque chose d'un peu plus impliqué, nous allons changer l'URL de la page job en quelque chose de plus significatif :

```
/job/sensio-labs/paris-france/1/web-developer
```

Sans rien savoir de Jobeet, et sans regarder la page, vous pouvez comprendre à partir de l'URL que Sensio Labs est à la recherche d'un développeur Web pour travailler à Paris en France.

Les URLs claires sont importantes car elles donnent des informations à l'utilisateur. C'est aussi très utile lorsqu'on copie l'URL dans un email ou pour optimiser le référencement de votre site sur les moteurs de recherche.

Le modèle suivant correpond à une telle URL :

```
/job/:company/:location/:id/:position
```

Editez le fichier `routing.yml` et ajouter la route `job_show_user` au début du fichier :

```
job_show_user:
  url:   /job/:company/:location/:id/:position
  param: { module: job, action: show }
```

Si vous actualisez la page d'accueil de Jobeet, les liens des emplois n'ont pas changé. C'est parce que pour générer une route, vous devez indiquer toutes les variables requises. Il faut donc modifier l'appel de `url_for()` dans `indexSuccess.php` par :

```
url_for('job/show?id='.$job->getId().'&company='.$job->getCompany().
  '&location='.$job->getLocation().'&position='.$job->getPosition())
```

Une URI interne peut aussi être définie dans un tableau :

```
url_for(array(
  'module'   => 'job',
  'action'   => 'show',
  'id'       => $job->getId(),
  'company'  => $job->getCompany(),
  'location' => $job->getLocation(),
  'position' => $job->getPosition(),
))
```

Requirements

Dans le tutoriel du premier jour, nous avons parlé de la validation et de la gestion d'erreur qui sont indispensables. Le routage possède son propre système de validation. Chaque modèle de variable peut être validé par une expression régulière définie en utilisant l'entrée requirements avec une définition de la route :

```
job_show_user:
  url:   /job/:company/:location/:id/:position
  param: { module: job, action: show }
  requirements:
    id: \d+
```

L'entrée `requirements` précédente requiert que `id` soit une valeur numérique. Si ce n'est pas le cas, la route ne fonctionnera pas.

La classe de la route

Chaque route définie dans `routing.yml` est convertie en objet de la classe `sfRoute`. Cette classe peut être modifiée en définissant une entrée `class` dans la définition de la route. Si le protocole HTTP vous est familier, vous savez qu'il définit plusieures "méthodes", `GET`, `POST`,

HEAD, DELETE, et PUT. Les trois premières sont supportées par tous les navigateurs, mais pas les deux autres.

Pour restreindre la route à une méthode déterminée, vous pouvez changer la classe de la route par sfRequestRoute et ajouter une condition pour la variable virtuelle sf_method :

```
job_show_user:
  url:   /job/:company/:location/:id/:position
  class: sfRequestRoute
  param: { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [get]
```

Exiger qu'une route corresponde uniquement à une des méthodes HTTP n'est pas totalement équivalent à l'utilisation de sfWebRequest::isMethod() dans vos actions. En effet le routage continuera de chercher une correspondance si la méthode ne correspond pas à celle définie.

Objet de la classe de la route

La nouvelle URI interne pour un emploi est un peu longue et pénible à écrire (url_for('job/show?id='.$job->getId().'&company='.$job->getCompany().'&location='.$job->getLocation().'&position='.$job->getPosition())). Mais comme nous venons de le voir dans la section précédente, la classe de la route peut être modifiée. Pour la route job_show_user, il est préférable d'utiliser la classe sfDoctrineRoute optimisée pour les routes représentées par des objets Doctrine ou des collections d'objets Doctrine :

```
job_show_user:
  url:     /job/:company/:location/:id/:position
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [get]
```

L'entrée options personnalise le comportement de la route. Ici, l'option model indique que la classe de modèle Doctrine (JobeetJob) est liée à la route et l'option type indique que cette route est attachée à un objet (vous pouvez aussi utiliser l'option list si une route représente une collection d'objets).

La route job_show_user sait maintenant qu'elle est liée avec JobeetJob et nous pouvons simplifier l'appel de url_for()url_for() :

```
url_for(array('sf_route' => 'job_show_user', 'sf_subject' => $job))
```

ou simplement :

```
url_for('job_show_user', $job)
```

Le premier exemple est utile quand vous devez passer plus d'arguments plutôt que l'objet seul.

Ca fonctionne car toutes les variables dans la route ont un accesseur correspondant dans la classe JobeetJob (par exemple, la variable company de la route est remplacée par la valeur de getCompany()).

A ce stade, les URLs générées ne sont pas encore comme nous le voudrions :

```
http://www.jobeet.com.localhost/frontend_dev.php/job/Sensio+Labs/Paris%2C+France/1/Web+
```

Nous devons utiliser un "slug" sur la valeur de la colonne pour remplaçer tous les charactères non ASCII par un -. Ouvrez le fichier JobeetJob et ajouter les méthodes suivantes à la classe :

```
// lib/model/doctrine/JobeetJob.class.php
public function getCompanySlug()
{
  return Jobeet::slugify($this->getCompany());
```

```
}

public function getPositionSlug()
{
  return Jobeet::slugify($this->getPosition());
}

public function getLocationSlug()
{
  return Jobeet::slugify($this->getLocation());
}
```

Ensuite, créer le fichier `lib/Jobeet.class.php` et y ajouter la fonction `slugify` :

```
// lib/Jobeet.class.php
class Jobeet
{
  static public function slugify($text)
  {
    // replace all non letters or digits by -
    $text = preg_replace('/\W+/', '-', $text);

    // trim and lowercase
    $text = strtolower(trim($text, '-'));

    return $text;
  }
}
```

Dans ce tutoriel, nous ne voyons jamais l'instruction d'ouverture <?php dans les exemples de code qui contiennent uniquement du pur code PHP pour optimiser l'espace et sauver un peu d'arbre. Vous devez évidemment ne pas oubliez de l'ajouter chaque fois que vous créez un nouveau fichier PHP. Rappelez-vous juste de ne pas l'ajouter à des fichiers Templates.

Nous venons de définir trois nouveaux accésseurs "virtuels" : `getCompanySlug()`, `getPositionSlug()` et `getLocationSlug()`. Ils retournent la valeur correspondante à leur colonne après avoir appliqué la méthode `slugify()`. Maintenant, vous pouvez remplaçer le nom réel des colonnes par leur équivalent virtuel dans la route `job_show_user` :

```
job_show_user:
  url:     /job/:company_slug/:location_slug/:id/:position_slug
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [get]
```

Les nouvelles URLs sont maintenant fonctionnelles :

`http://www.jobeet.com.localhost/frontend_dev.php/job/sensio-labs/paris-france/1/web-developer`

Mais nous ne sommes qu'au milieu de l'histoire. La route est capable de générer une URL basée sur un objet, mais elle peut aussi trouver l'objet lié à une URL donnée. L'objet peut être récupéré avec la méthode `getObject()` de l'objet de la route. A l'analyse d'une requête entrante, le routage stocke la route correspondante en un objet qui peut être utilisé dans les actions. Donc, modifiez la méthode `executeShow()` pour utiliser l'objet de la route et retrouver l'objet `Jobeet` :

```
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();

    $this->forward404Unless($this->job);
  }

  // ...
```

```
}
```

Si vous essayez d'afficher un job avec un id inconnu, vous verrez la page d'erreur 404 mais avec un message différent :

C'est parce que l'erreur 404 a été levée automatiquement par la méthode getRoute(). Donc, nous pouvons encore simplifier la méthode executeShow :

```
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();
  }

  // ...
}
```

Si vous ne voulez pas que la route génère une erreur 404, vous pouvez paramétrer l'option de routage allow_empty sur true.

L'objet associé d'une route n'est pas chargé. Il est seulement récupéré de la base de données si vous appelez la méthode getRoute().

Le routage dans les Actions et les Templates

Dans un template, le helper url_for() convertit une URI interne en une URL externe. D'autres helpers de symfony utilisent aussi une URI interne en tant qu'argument, comme le helper link_to() qui génère une balise <a> :

```
<?php echo link_to($job->getPosition(), 'job_show_user', $job) ?>
```

Il produit le code HTML suivant :

```
<a href="/job/sensio-labs/paris-france/1/web-developer">Web Developer</a>
```

Tous les deux, url_for() et link_to() peuvent aussi générer des URLs absolues :

```
url_for('job_show_user', $job, true);

link_to($job->getPosition(), 'job_show_user', $job, true);
```

Si vous voulez générer une URL depuis une action, vous pouvez utiliser la méthode generateUrl() :

```
$this->redirect($this->generateUrl('job_show_user', $job));
```

Les méthodes de la famille "redirect"

Dans le tutorial d'hier, nous avons parlé des méthodes "forward". Ces méthodes renvoient la requête en cours vers une autre action sans rechargement de la page avec le navigateur.

Les méthodes "redirect" redirigent l'utilisateur vers une autre URL. Comme pour forward, vous pouvez utiliser la méthode redirect() ou les méthodes de raccourci redirectIf() et redirectUnless().

La classe Collection de la route

Pour le module `job`, nous avons déjà personnalisé la route pour l'action `show`. Mais les URLs pour les autres méthodes (`index`, `new`, `edit`, `create`, `update`, et `delete`) sont encore gérées par la route `default` :

```
default:
  url: /:module/:action/*
```

La route `default` est idéale pour commençer à coder sans avoir besoin de définir beaucoup de routes. Mais comme cette route agit comme un "fourre-tout", elle ne peut pas être configurée pour des besoins spécifiques.

Toutes les actions pour `job` sont liées à la classe de modèle `JobeetJob`. Nous pouvons facilement définir une route `sfDoctrineRoute` personnalisée pour chaque action comme cela a été déjà fait pour l'action ~show. Etant donné que le module `job` définit déjà les septs actions classiques pour un modèle, nous pouvons aussi utiliser la classe `sfDoctrineRouteCollection`. Ouvrez le fichier `routing.yml` et modifiez le comme suit :

```
# apps/frontend/config/routing.yml
job:
  class:   sfDoctrineRouteCollection
  options: { model: JobeetJob }

job_show_user:
  url:     /job/:company_slug/:location_slug/:id/:position_slug
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [get]

# default rules
homepage:
  url:   /
  param: { module: job, action: index }

default_index:
  url:   /:module
  param: { action: index }

default:
  url:   /:module/:action/*
```

La route `job` ci-dessus est tout simplement un raccourci qui génère automatiquement les sept routes `sfDoctrineRoute` suivantes :

```
job:
  url:     /job.:sf_format
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: list }
  param:   { module: job, action: index, sf_format: html }
  requirements: { sf_method: get }

job_new:
  url:     /job/new.:sf_format
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: new, sf_format: html }
  requirements: { sf_method: get }

job_create:
  url:     /job.:sf_format
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: create, sf_format: html }
  requirements: { sf_method: post }

job_edit:
```

```
  url:     /job/:id/edit.:sf_format
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: edit, sf_format: html }
  requirements: { sf_method: get }

job_update:
  url:     /job/:id.:sf_format
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: update, sf_format: html }
  requirements: { sf_method: put }

job_delete:
  url:     /job/:id.:sf_format
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: delete, sf_format: html }
  requirements: { sf_method: delete }

job_show:
  url:     /job/:id.:sf_format
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show, sf_format: html }
  requirements: { sf_method: get }
```

Quelques routes générées par `sfDoctrineRouteCollection` ont la même URL. Le routage est capable de les utiliser correctement car elles ont toutes une méthode HTTP différentes dans l'entrée requirements.

Les routes `job_delete` et `job_update` requièrent des méthodes HTTP non supportées par les navigateurs (respectivement DELETE et PUT). Ca fonctionne car symfony les simulent. Pour voir un exemple, ouvrez le Template `_form.php` :

```
// apps/frontend/modules/job/templates/_form.php
<form action="..." ...>
<?php if (!$form->getObject()->isNew()): ?>
  <input type="hidden" name="sf_method" value="PUT" />
<?php endif; ?>

<?php echo link_to(
  'Delete',
  'job/delete?id='.$form->getObject()->getId(),
  array('method' => 'delete', 'confirm' => 'Are you sure?')
) ?>
```

Tous les helpers symfony peuvent être utilisés pour simuler n'importe quelle méthode HTTP grâce au paramètre spécial `sf_method`.

symfony possède d'autres paramètres spéciaux comme `sf_method`, qui débutent tous par le préfixe `sf_`. Dans les routes générées ci-dessus, vous pouvez en voir un autre : `sf_format` qui sera expliqué un autre jour.

Débogage d'une route

Quand vous avez beaucoup de routes, il est parfois utile de lister les routes générées. La tâche `app:routes` affiche toutes les routes d'une application donnée :

```
$ php symfony app:routes frontend
```

Il est également possible d'obtenir des informations de debogage en passant le nom de la route en argument :

```
$ php symfony app:routes frontend job_edit
```

Routes par défaut

Définir des routes pour toutes les URLs est une bonne méthode de travail. Puisque la route `job`

définit toutes les routes nécessaires pour l'utilisation de l'application Jobeet, nous pouvons supprimer ou commenter les routes par défaut dans le fichier de configuration `routing.yml` :

```
# apps/frontend/config/routing.yml
#default_index:
#  url:   /:module
#  param: { action: index }
#
#default:
#  url:   /:module/:action/*
```

L'application Jobeet fonctionne toujours à l'identique.

Conclusion

Ce chapitre a introduit beaucoup de nouvelles informations. Vous avez appris à utiliser le framework de routage de symfony et à dissocier vos URLs de la réalisation technique.

Au cours du chapitre suivant, nous n'introduirons aucun concept nouveau, mais nous passerons davantage de temps à étudier plus en détails tout ce que nous avons découvert jusqu'à présent.

« Jour 4 : Le contrôleur et la vue

Jour 6 : Aller plus loin avec le Modèle »

Questions & Feedback

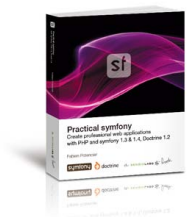
If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 6 : Aller plus loin avec le Modèle

Hier était un grand jour. Vous avez appris comment créer des URLs propres et comment utiliser le framework symfony pour automatiser beaucoup de choses pour vous.

Aujourd'hui, nous allons améliorer Jobeet en optimisant le code ci et là. Vous en apprendrez plus sur toutes les fonctions que nous avons déjà présenté dans ce tutoriel au cours des jours précédents.



L'objet Query de Doctrine

Conditions du Jour 2 :

"Quand un utilisateur arrive sur Jobeet, il doit voir la liste des emplois actifs."

Mais pour l'instant, tous les emplois sont affichés, qu'ils soient actifs ou non :

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeIndex(sfWebRequest $request)
  {
    $this->jobeet_jobs = Doctrine::getTable('JobeetJob')
      ->createQuery('a')
      ->execute();
  }

  // ...
}
```

Un emploit est considéré actif s'il a été posté il y a moins de 30 jours. La méthode ~Doctrine_Query~::execute() crée la requête à exécuter sur la base de données. Dans le code ci-dessus, aucune condition n'est spécifiée, ce qui signifie que tous les enregistrements seront retournés.

Modifions cela pour n'afficher que les emplois actifs :

```
public function executeIndex(sfWebRequest $request)
{
  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.created_at > ?', date('Y-m-d H:i:s', time() - 86400 * 30));

  $this->jobeet_jobs = $q->execute();
}
```

Débogage du SQL généré par Doctrine

Etant donné que vous n'écrivez pas les requêtes SQL à la main, Doctrine se chargera de vous les différencier entre les moteurs de base de données et générera les instructions SQL optimisé pour le moteur de base choisi pendant la journée 3. Mais parfois, voir le SQL généré par Doctrine est d'une grande aide, par exemple, pour déboguer une requête qui ne fonctionne pas comme prévu. Dans l'environnement de dev, symfony journalise ces requêtes (et plus encore) dans le répertoire log/. Il existe un fichier log pour chaque couple application/environnement. Le fichier que nous recherchons est nommé frontend_dev.log:

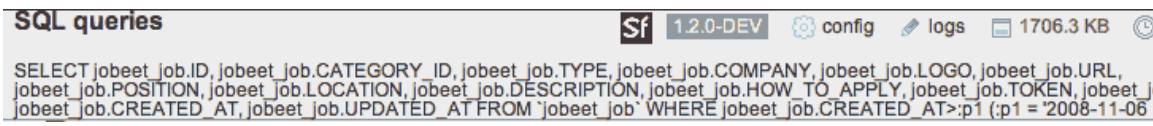
```
# log/frontend_dev.log
Dec 04 13:58:33 symfony [info] {sfDoctrineLogger} executeQuery : SELECT
j.id AS j__id, j.category_id AS j__category_id, j.type AS j__type,
j.company AS j__company, j.logo AS j__logo, j.url AS j__url,
j.position AS j__position, j.location AS j__location,
```

```
j.description AS j__description, j.how_to_apply AS j__how_to_apply,
j.token AS j__token, j.is_public AS j__is_public,
j.is_activated AS j__is_activated, j.email AS j__email,
j.expires_at AS j__expires_at, j.created_at AS j__created_at,
j.updated_at AS j__updated_at FROM jobeet_job j
WHERE j.created_at > ? (2008-11-08 01:13:35)
```

Comme vous pouvez le voir, Doctrine a généré une clause WHERE pour la colonne `created_at` (`WHERE j.created_at > ?`).

La chaine ? dans la requête indique que Doctrine génère les instructions préparées. La valeur actuelle de ? ('2008-11-08 01:13:35' dans l'exemple ci-dessus) est passée au cours de l'exécution de la requête et elle est correctement échappée par le moteur de base de données. L'utilisation d'instructions préparées réduit considérablement votre exposition aux attaques par injection SQL.

Le travail est facilité mais devoir basculer entre le navigateur, l'IDE et le fichier log à chaque fois que l'on veut tester une modification est assez contraignant. Heureusement, symfony possède une barre d'outil de débogage. Toutes les informations nécessaire sont disponibles dans votre navigateur :



Sérialisation d'un objet

Jusqu'à présent, notre code fonctionne mais il est loin d'être parfait et ne prend pas en charge les contraintes évoqués le 2ème jour :

"Un utilisateur peut activer à nouveau ou augmenter la validité de l'offre d'emploi pour une période de 30 jours supplémentaires..."

Le code actuel se base sur la valeur de la colonne `created_at` qui stocke la date de création ce qui ne nous permet pas de satisfaire la condition ci-dessus.

Mais si vous vous rappelez le schéma de la base de données décrit le 3ème jour, nous avons aussi défini une colonne `expires_at`. Pour l'instant cette valeur est vide car nous ne l'avons pas renseignée dans le fichier de jeu de test (fixture). Mais lorsqu'un emploi est créé, elle peut être automatiquement renseignée 30 jours plus tard par rapport à la date courante.

Quand vous devez créer une action automatique avant que l'objet Doctrine soit sérialisé dans la base, vous pouvez surcharger la méthode `save()` de la classe du modèle :

```
// lib/model/doctrine/JobeetJob.class.php
class JobeetJob extends BaseJobeetJob
{
  public function save(Doctrine_Connection $conn = null)
  {
    if ($this->isNew() && !$this->getExpiresAt())
    {
      $now = $this->getCreatedAt() ? $this->getDateTimeObject('created_at')-
>format('U') : time();
      $this->setExpiresAt(date('Y-m-d H:i:s', $now + 86400 * 30));
    }

    return parent::save($conn);
  }

  // ...
}
```

La méthode `isNew()` renvoie `true` quand l'objet n'est pas encore sérialisé dans la base et `false` dans la cas contraire.

A présent, modifions l'action pour récupérer les emplois actifs en utilisant la colonne `expires_at` au lieu de la colonne `created_at` :

```
public function executeIndex(sfWebRequest $request)
{
  $q = Doctrine_Query::create()
```

```
    ->from('JobeetJob j')
    ->where('j.expires_at > ?', date('Y-m-d H:i:s', time()));

  $this->jobeet_jobs = $q->execute();
}
```

La requête sélectionnera seulement les emplois possédant une date `expires_at` dans le future.

Aller plus loin avec les jeux de test

L'actualisation de la page d'accueil Jobeet dans votre navigateur ne va rien changer car les emplois dans la base de données ont été posté il y a tout juste quelques jours. Changeons les jeux de test (fixture) pour ajouter une tâche qui a déjà expiré :

```
# data/fixtures/jobs.yml
JobeetJob:
  # other jobs

  expired_job:
    JobeetCategory: programming
    company:        Sensio Labs
    position:       Web Developer
    location:       Paris, France
    description:    Lorem ipsum dolor sit amet, consectetur adipisicing elit.
    how_to_apply:   Send your resume to lorem.ipsum [at] dolor.sit
    is_public:      true
    is_activated:   true
    created_at:     '2005-12-01 00:00:00'
    token:          job_expired
    email:          job@example.com
```

Faites bien attention quand vous faîtes un copier/coller du code dans le fichier fixture|Jeux de test. Il faut conserver l'indentation. Il doit y avoir deux espaces devant `expired_job`.

Comme vous pouvez le constater, il est possible de définir une valeur pour la colonne `created_at` même si elle est automatiquement remplie par Doctrine. La valeur définie sera utilisée à la place de la valeur automatique. Rechargez les jeux de test et actualisez la page d'accueil pour vérifier que l'ancien emploi n'apparaisse pas :

```
$ php symfony doctrine:data-load
```

Vous pouvez aussi exécuter la requête suivante pour être sûr que la colonne `expires_at` soit automatiquement renseignée en fonction de la valeur de la colonne `created_at` grâce à la méthode `save()` :

```
SELECT `position`, `created_at`, `expires_at` FROM `jobeet_job`;
```

Configuration personnalisée

Dans la méthode `JobeetJob::save()`, nous avons figé le nombre de jours qui détermine l'expiration d'un emploi. Il serait préférable que la valeur de 30 jours soit paramétrable. Le framework symfony utilise le fichier de configuration interne `app.yml` qui permet de définir des paramètres|Paramètres spécifiques à l'application|Application. Ce fichier YAML peut contenir n'importe quel paramètre nécessaire :

```
# apps/frontend/config/app.yml
all:
  active_days: 30
```

Dans l'application, ces paramètres sont disponibles à travers la classe globale `sfConfig` :

```
sfConfig::get('app_active_days')
```

Les paramètres utilisent le préfixe `app_` car la classe `sfConfig` fournit également des accès aux paramètres symfony que nous verrons plus tard.

Mettez le code à jour pour prendre en compte ce nouveau paramètre :

```
public function save(Doctrine_Connection $conn = null)
{
  if ($this->isNew() && !$this->getExpiresAt())
```

```
  {
    $now = $this->getCreatedAt() ? $this->getDateTimeObject('created_at')->format('U')
: time();
    $this->setExpiresAt(date('Y-m-d H:i:s', $now + 86400 *
sfConfig::get('app_active_days')));
  }

  return parent::save($conn);
}
```

Le fichier de configuration `app.yml` est un bon moyen de centraliser les paramètres globaux|Paramètres globaux de votre application.

Pour finir, si vous avez besoin de définir des paramètres étendus|Configuration globale, il suffit de créer un nouveau fichier `app.yml` dans le répertoire `config` à la racine de votre projet symfony.

Refactorisation

Bien que notre code fonctionne correctement, il n'est pas encore parfait. Etes-vous capable de repérer le problème ?

Le code `Doctrine_Query` n'appartient pas à l'action (la couche Contrôleur), mais à la couche Modèle. Dans le modèle MVC, le Modèle définit toute la logique métier|Logique métier, et le Controlleur appelle le Modèle pour récupérer les données. Etant donné que le code renvoie une collection d'emplois, déplaçons le dans la classe `JobeetJobTable` et créons la méthode `getActiveJobs()` :

```
// lib/model/doctrine/JobeetJobTable.class.php
class JobeetJobTable extends Doctrine_Table
{
  public function getActiveJobs()
  {
    $q = $this->createQuery('j')
      ->where('j.expires_at > ?', date('Y-m-d H:i:s', time()));

    return $q->execute();
  }
}
```

Maintenant le code de l'action peut utiliser cette nouvelle méthode pour récupérer les emplois actifs.

```
public function executeIndex(sfWebRequest $request)
{
  $this->jobeet_jobs = Doctrine_Core::getTable('JobeetJob')->getActiveJobs();
}
```

La refactorisation|Refactorisation a plusieurs avantages par rapport au code précédent :

- La logique pour obtenir les emplois actifs est maintenant dans le modèle, la où est sa place
- Le code du Contrôleur est plus lisible
- La méthode `getActiveJobs()` est réutilisable (dans une autre action par exemple)
- Le code modèle est désormais testable indépendament

Récupérons les emplois grâce à la colonne `expires_at` :

```
public function getActiveJobs()
{
  $q = $this->createQuery('j')
    ->where('j.expires_at > ?', date('Y-m-d H:i:s', time()))
    ->orderBy('j.expires_at DESC');

  return $q->execute();
}
```

La méthode `orderBy` ajoute une clause `ORDER BY` à la requête. (`addOrderBy()` existe aussi).

Catégories en page d'accueil

Conditions du 2ème jour :

"Les emplois sont affichés par catégorie et par date de publication (les nouveaux emplois en tête

de liste)."

Jusqu'à présent, nous n'avons pas pris en compte la catégorie associée aux emplois. Afin d'afficher les emplois par catégorie, nous allons d'abord récupérer toutes les catégories associées à au moins un emploi.

Editez la classe `JobeetCategoryTable` et ajoutez la méthode `getWithJobs()` :

```
// lib/model/doctrine/JobeetCategoryTable.class.php
class JobeetCategoryTable extends Doctrine_Table
{
  public function getWithJobs()
  {
    $q = $this->createQuery('c')
      ->leftJoin('c.JobeetJobs j')
      ->where('j.expires_at > ?', date('Y-m-d H:i:s', time()));

    return $q->execute();
  }
}
```

Modifiez l'action `index` en conséquence :

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeIndex(sfWebRequest $request)
{
  $this->categories = Doctrine_Core::getTable('JobeetCategory')->getWithJobs();
}
```

Dans le Template, nous devons rechercher les emplois actifs dans chaque catégorie et les afficher.

```
// apps/frontend/modules/job/templates/indexSuccess.php
<?php use_stylesheet('jobs.css') ?>

<div id="jobs">
  <?php foreach ($categories as $category): ?>
    <div class="category_<?php echo Jobeet::slugify($category->getName()) ?>">
      <div class="category">
        <div class="feed">
          <a href="">Feed</a>
        </div>
        <h1><?php echo $category ?></h1>
      </div>

      <table class="jobs">
        <?php foreach ($category->getActiveJobs() as $i => $job): ?>
          <tr class="<?php echo fmod($i, 2) ? 'even' : 'odd' ?>">
            <td class="location">
              <?php echo $job->getLocation() ?>
            </td>
            <td class="position">
              <?php echo link_to($job->getPosition(), 'job_show_user', $job) ?>
            </td>
            <td class="company">
              <?php echo $job->getCompany() ?>
            </td>
          </tr>
        <?php endforeach; ?>
      </table>
    </div>
  <?php endforeach; ?>
</div>
```

Pour afficher le nom d'une catégorie, nous utilisons `echo $category` dans le Template. Ça vous paraît bizarre ? `$category` est un objet, comment peut-on afficher de façon magique le nom de la catégorie avec un echo. La réponse se trouve au jour 3 quand nous avons défini la méthode magique `__toString()` pour toutes les classes du modèle.

Pour que cela fonctionne, nous devons ajouter la méthode `getActiveJobs()` à la classe `JobeetCategory` :

```
// lib/model/doctrine/JobeetCategory.class.php
public function getActiveJobs()
{
  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.category_id = ?', $this->getId());

  return Doctrine_Core::getTable('JobeetJob')->getActiveJobs($q);
}
```

La méthode `JobeetCategory::getActiveJobs()` utilise la méthode `Doctrine::getTable('JobeetJob')->getActiveJobs()` pour rechercher les emplois actifs de la catégorie donnée.

A l'appel de `Doctrine::getTable('JobeetJob')->getActiveJobs()`, nous voulons restreindre la condition autrement qu'en fournissant uniquement une catégorie. Au lieu de passer l'objet catégory, nous avons décidé de passer un objet `Doctrine_Query` qui est la meilleure solution pour encapsuler une condition générique.

Pour ce faire, il faut fusionner cet objet `Doctrine_Query` avec les critères de la méthode `getActiveJobs()`. `Doctrine_Query` étant un objet, ce sera simple :

```
// lib/model/doctrine/JobeetJobTable.class.php
public function getActiveJobs(Doctrine_Query $q = null)
{
  if (is_null($q))
  {
    $q = Doctrine_Query::create()
      ->from('JobeetJob j');
  }

  $q->andWhere('j.expires_at > ?', date('Y-m-d H:i:s', time()))
    ->addOrderBy('j.expires_at DESC');

  return $q->execute();
}
```

Limiter les résultats

Il reste encore une condition à implémenter pour la liste des emplois en page d'accueil :

"Chaque catégorie doit afficher les 10 premiers emplois et un lien doit permettre d'afficher tous les emplois d'une catégorie choisie."

C'est assez simple de l'ajouter à la méthode `getActiveJobs()` :

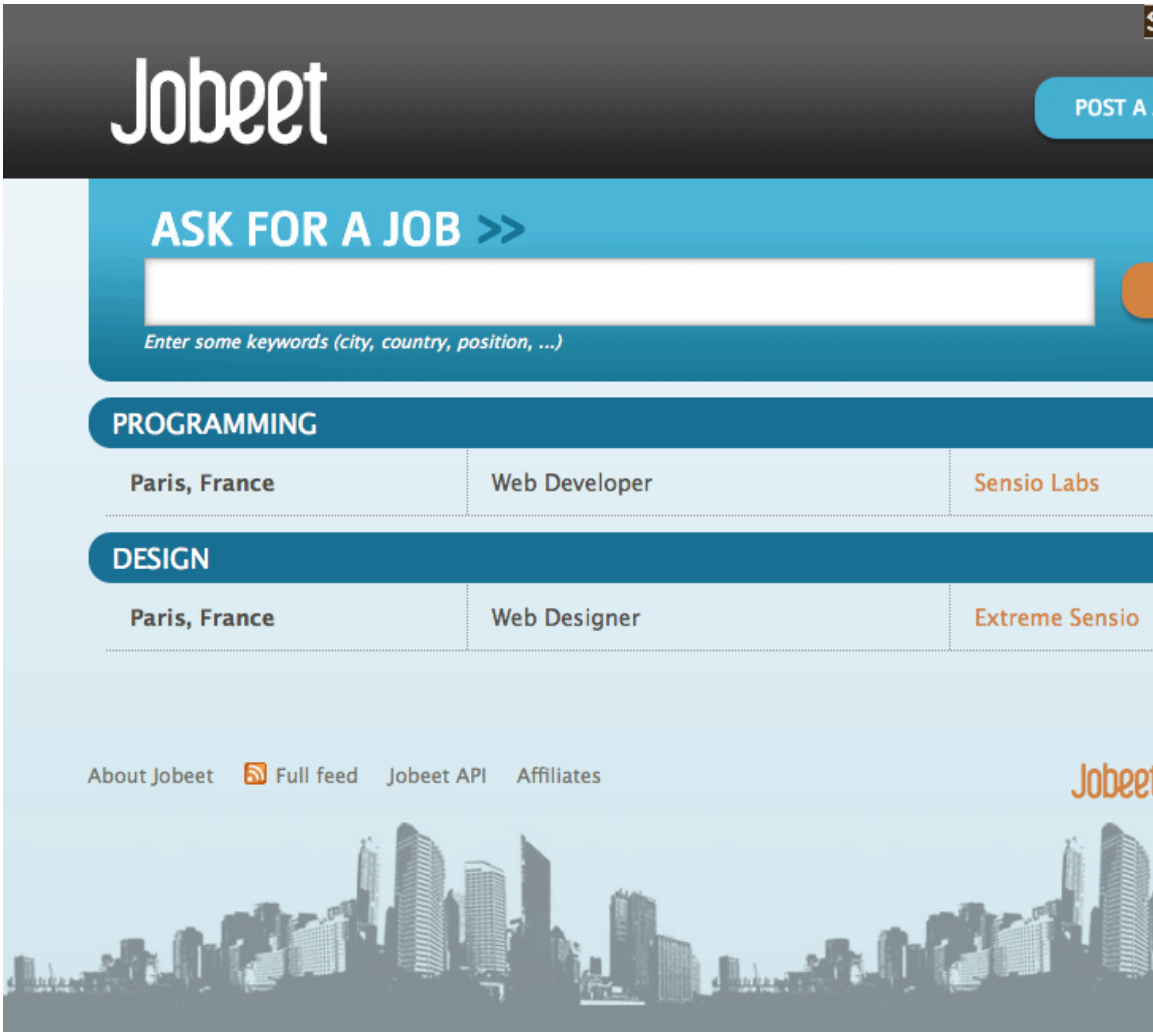
```
// lib/model/doctrine/JobeetCategory.class.php
public function getActiveJobs($max = 10)
{
  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.category_id = ?', $this->getId())
    ->limit($max);

  return Doctrine_Core::getTable('JobeetJob')->getActiveJobs($q);
}
```

La clause `LIMIT` est codée en dur dans le Modèle, mais il est préférable de pouvoir configurer cette valeur. Modifiez le Template pour utiliser le nombre maximum d'emplois configuré dans `app.yml` :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<?php foreach ($category->getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')) as
$i => $job): ?>
```

et ajoutez le nouveau paramètre dans `app.yml` :

```
all:
  active_days:          30
  max_jobs_on_homepage: 10
```



Jeux de test dynamiques

A moins de passer la valeur `max_jobs_on_homepage` à un, vous ne verrez aucune différence. Nous devons ajouter des emplois dans le fichier fixture|Fixtures. Évidemment, vous pouvez faire 20, 30, ... copier/coller des emplois existants mais il y a une meilleure solution. La duplication n'est pas une bonne méthode, même pour les fichiers fixture.

symfony à la rescousse ! Dans symfony, les fichiers YAML peuvent contenir du code PHP qui sera évalué juste avant l'analyse du fichier. Editez le fichier fixture `jobs.yml` et ajoutez le code suivant à la fin :

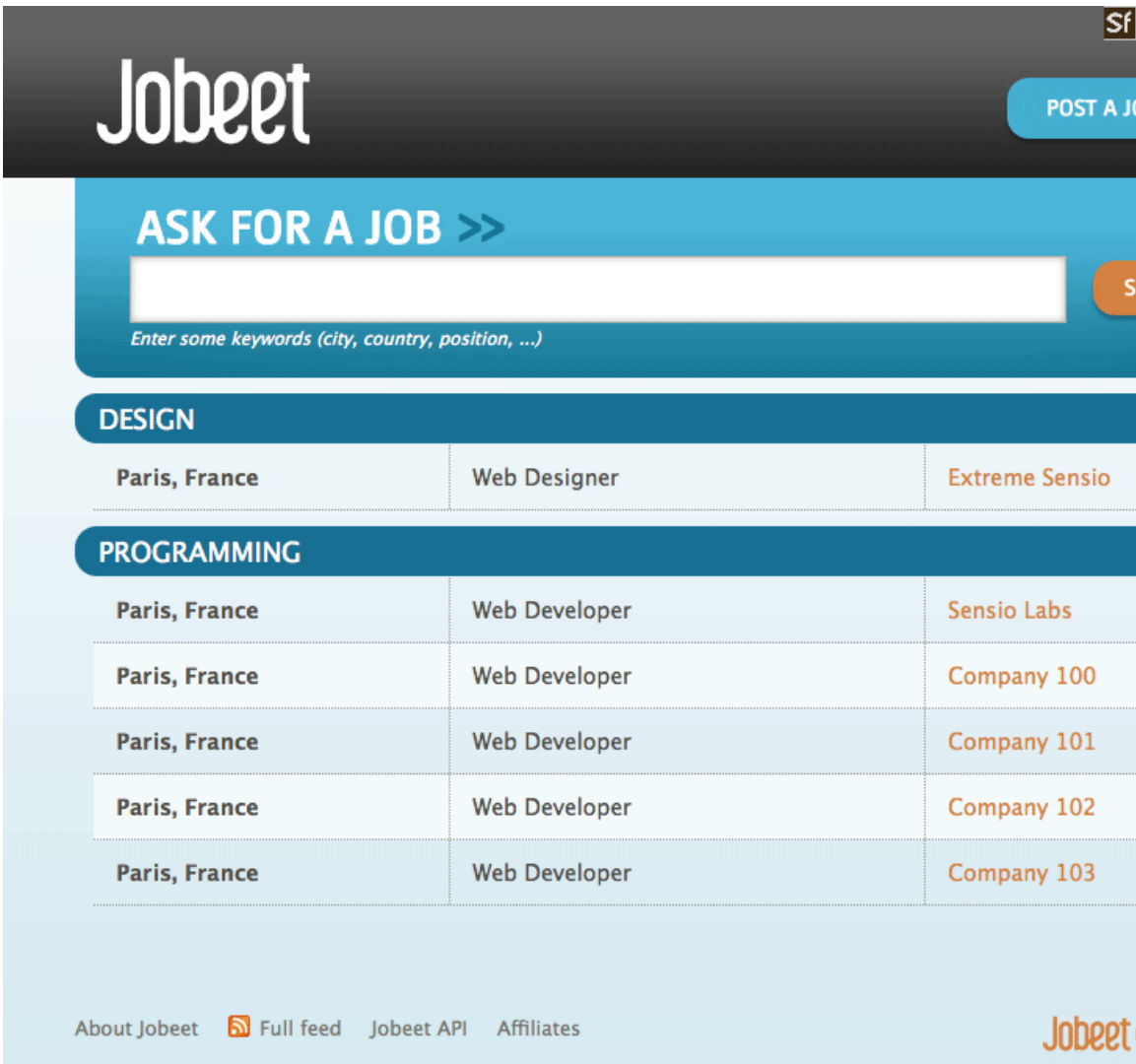
```
# Démarrez au début de la ligne (pas d'espace avant)
<?php for ($i = 100; $i <= 130; $i++): ?>
  job_<?php echo $i ?>:
    JobeetCategory: programming
    company:      Company <?php echo $i."\n" ?>
    position:     Web Developer
    location:     Paris, France
    description:  Lorem ipsum dolor sit amet, consectetur adipisicing elit.
    how_to_apply: |
      Send your resume to lorem.ipsum [at] company_<?php echo $i ?>.sit
    is_public:    true
    is_activated: true
    token:        job_<?php echo $i."\n" ?>
    email:        job@example.com

<?php endfor ?>
```

Attention ! L'analyseur de YAML n'aime pas les erreurs d' indentation|Formatage du code. Gardez bien à l'esprit les conseils suivants si vous ajoutez du code PHP dans un fichier YAML :

- L'instruction <?php ?> oit toujours commencer une ligne ou être intégré dans une valeur.
- Si l'instructione <?php ?> termine une ligne, vous devez indiquer clairement une nouvelle

ligne ("\n").

Rechargez les jeux de test avec la tâche `doctrine:data-load` et vérifiez que seulement `10` offres d'emplois soient affichées en page d'accueil pour la catégorie `Programming`. Dans la capture d'écran suivante, nous avons diminué le nombre maximum d'offres à 5 afin d'obtenir une image de taille raisonnable :



Sécurisez la page emploi

Même si vous connaissez l'URL d'une offre qui a expiré, il ne doit plus être possible d'y accéder. Essayez l'URL d'un emploi expiré (remplacez l'id par l'id correspondant dans la base de donnée - `SELECT id, token FROM jobeet_job WHERE expires_at < NOW()`) :

```
/frontend_dev.php/job/sensio-labs/paris-france/ID/web-developer-expired
```

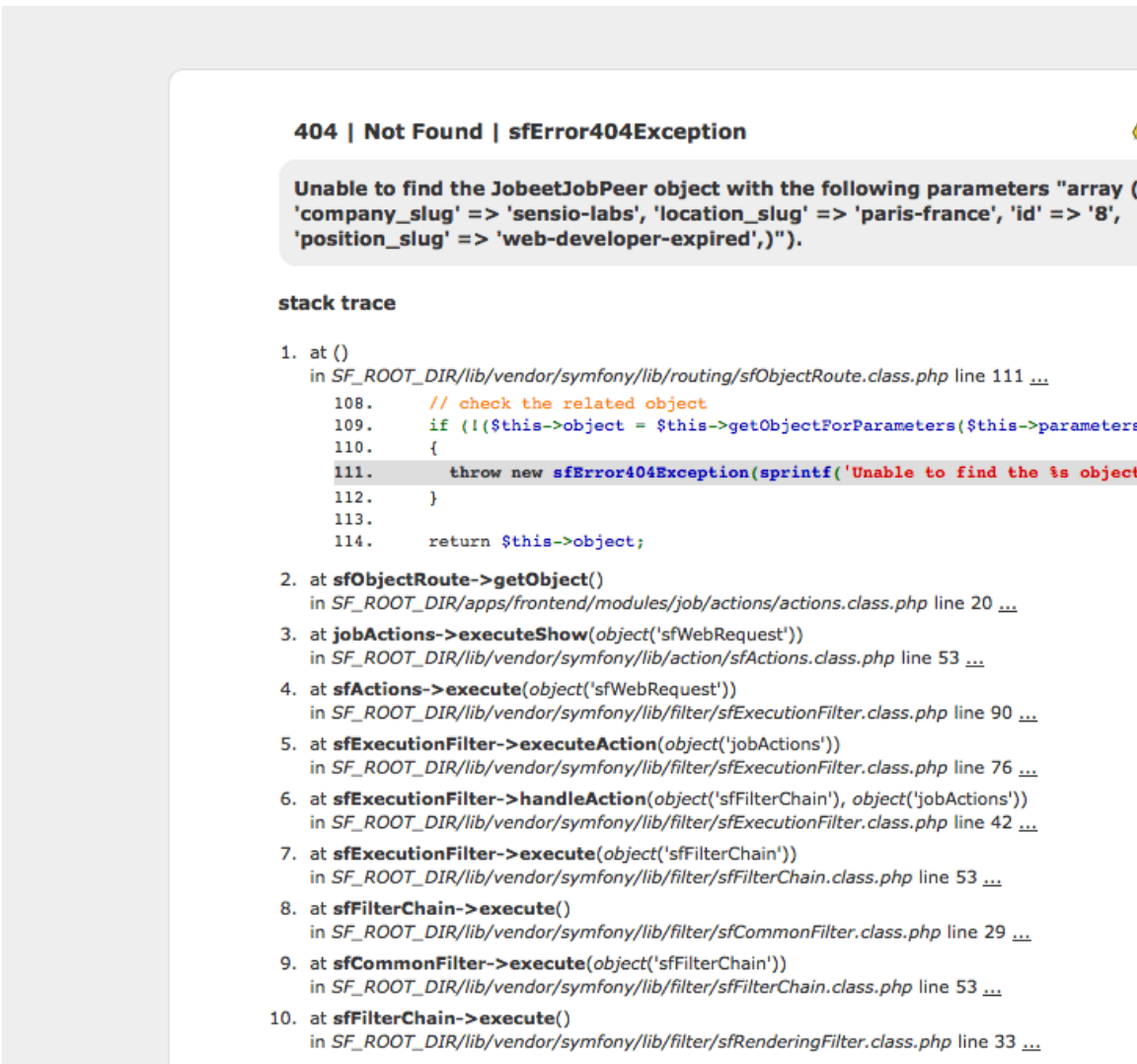
Au lieu d'afficher l'emploi, nous devons rediriger l'utilisateur vers une erreur 404. Mais comment faire alors que l'emploi est recherché automatiquement par la route ?

```
# apps/frontend/config/routing.yml
job_show_user:
  url:     /job/:company_slug/:location_slug/:id/:position_slug
  class:   sfDoctrineRoute
  options:
    model: JobeetJob
    type:  object
    method_for_query: retrieveActiveJob
  param:   { module: job, action: show }
  requirements:
    id: \d+
    sf_method: [GET]
```

La méthode `retrieveActiveJob()` recevra l'objet `Doctrine_Query` construit par la route :

```php
// lib/model/doctrine/JobeetJobTable.class.php
class JobeetJobTable extends Doctrine_Table
{
  public function retrieveActiveJob(Doctrine_Query $q)
  {
    $q->andWhere('a.expires_at > ?', date('Y-m-d H:i:s', time()));

    return $q->fetchOne();
  }

  // ...
}
```

Maintenant, si vous essayez d'obtenir un emploi expiré, vous serez envoyé sur une page 404.



Lien vers la page catégorie

A présent, nous allons ajouter un lien vers la page catégorie et créer la page catégorie.

Une minute. L'heure n'est pas encore écoulée et nous n'avons pas beaucoup travaillé. En fait, vous avez tout le temps nécessaire pour mettre en pratique tout ce que nous avons déjà appris et implémenter cette fonction par vous-même. Vous pourrez vérifier votre travail demain.

Conclusion

Prenez le temps de développer votre projet Jobeet en local. N'hésitez pas à abuser de la documentation en ligne de l'API et de toute la documentation|Documentation~ gratuite disponible sur le site pour vous aider. Le chapitre suivant donne la solution d'implémentation de cette fonctionnalité.

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

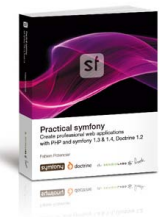
If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 7 : Jouons avec la page Catégorie

Hier, vous avez pu améliorer vos connaissances dans plusieurs domaines : requêtes avec Doctrine, jeux de test, routage, débogage et la configuration personnalisée. Et nous avions fini la leçon en vous laissant un petit défi pour débuter ce chapitre.

Nous espèrons que vous avez travaillé sur la page catégorie de Jobeet car le tutoriel d'aujourd'hui aura alors beaucoup plus de valeur pour vous.

Prêt ? Voici une des implémentations possibles.



La route de la catégorie

Pour commencer, nous devons ajouter une route pour utiliser des URL propres. Ajoutez ce code au début du fichier de configuration :

```
# apps/frontend/config/routing.yml
category:
  url:      /category/:slug
  class:    sfDoctrineRoute
  param:    { module: category, action: show }
  options:  { model: JobeetCategory, type: object }
```

A chaque fois que vous commencez à implémenter une nouvelle fonctionnalité, créer une route pour les URL en premier est une bonne méthode de travail. Et c'est obligatoire si jamais vous supprimez les règles de routage par défaut.

Une route peut avoir comme paramètre n'importe quelle colonne de l'objet associé. Il est également possible d'utiliser n'importe quelle autre valeur si un accesseur est défini dans la classe de l'objet. Étant donné que le paramètre `slug` ne fait référence à aucune colonne dans la table `category`, nous devons ajouter un accessseur virtuel dans `JobeetCategory` pour faire fonctionner la route :

```
// lib/model/doctrine/JobeetCategory.class.php
public function getSlug()
{
  return Jobeet::slugify($this->getName());
}
```

Le lien de la catégorie

A présent, éditez le Template `indexSuccess.php` du module `job` et ajoutez le lien vers la page catégorie :

```
<!-- some HTML code -->

        <h1>
          <?php echo link_to($category, 'category', $category) ?>
        </h1>

<!-- some HTML code -->

      </table>

      <?php if (($count = $category->countActiveJobs() -
sfConfig::get('app_max_jobs_on_homepage')) > 0): ?>
        <div class="more_jobs">
          and <?php echo link_to($count, 'category', $category) ?>
          more...
        </div>
```

```
      <?php endif; ?>
    </div>
  <?php endforeach; ?>
</div>
```

Nous ajoutons seulement les liens, s'il y a plus de 10 emplois à afficher pour la catégorie actuelle. Le lien contient le nombre d'emplois non affichés. Pour que ce Template fonctionne, nous devons ajouter la méthod `countActiveJobs()` pour `JobeetCategory` :

```
// lib/model/doctrine/JobeetCategory.class.php
public function countActiveJobs()
{
  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.category_id = ?', $this->getId());

  return Doctrine_Core::getTable('JobeetJob')->countActiveJobs($q);
}
```

La méthode `countActiveJobs()` utilise la méthode `countActiveJobs()` qui n'existe pas encore dans le modèle `JobeetJobTable`. Remplacer le contenu du fichier `JobeetJobTable.php` par le code suivant :

```
// lib/model/doctrine/JobeetJobTable.class.php
class JobeetJobTable extends Doctrine_Table
{
  public function retrieveActiveJob(Doctrine_Query $q)
  {
    return $this->addActiveJobsQuery($q)->fetchOne();
  }

  public function getActiveJobs(Doctrine_Query $q = null)
  {
    return $this->addActiveJobsQuery($q)->execute();
  }

  public function countActiveJobs(Doctrine_Query $q = null)
  {
    return $this->addActiveJobsQuery($q)->count();
  }

  public function addActiveJobsQuery(Doctrine_Query $q = null)
  {
    if (is_null($q))
    {
      $q = Doctrine_Query::create()
        ->from('JobeetJob j');
    }

    $alias = $q->getRootAlias();

    $q->andWhere($alias . '.expires_at > ?', date('Y-m-d H:i:s', time()))
      ->addOrderBy($alias . '.created_at DESC');

    return $q;
  }
}
```

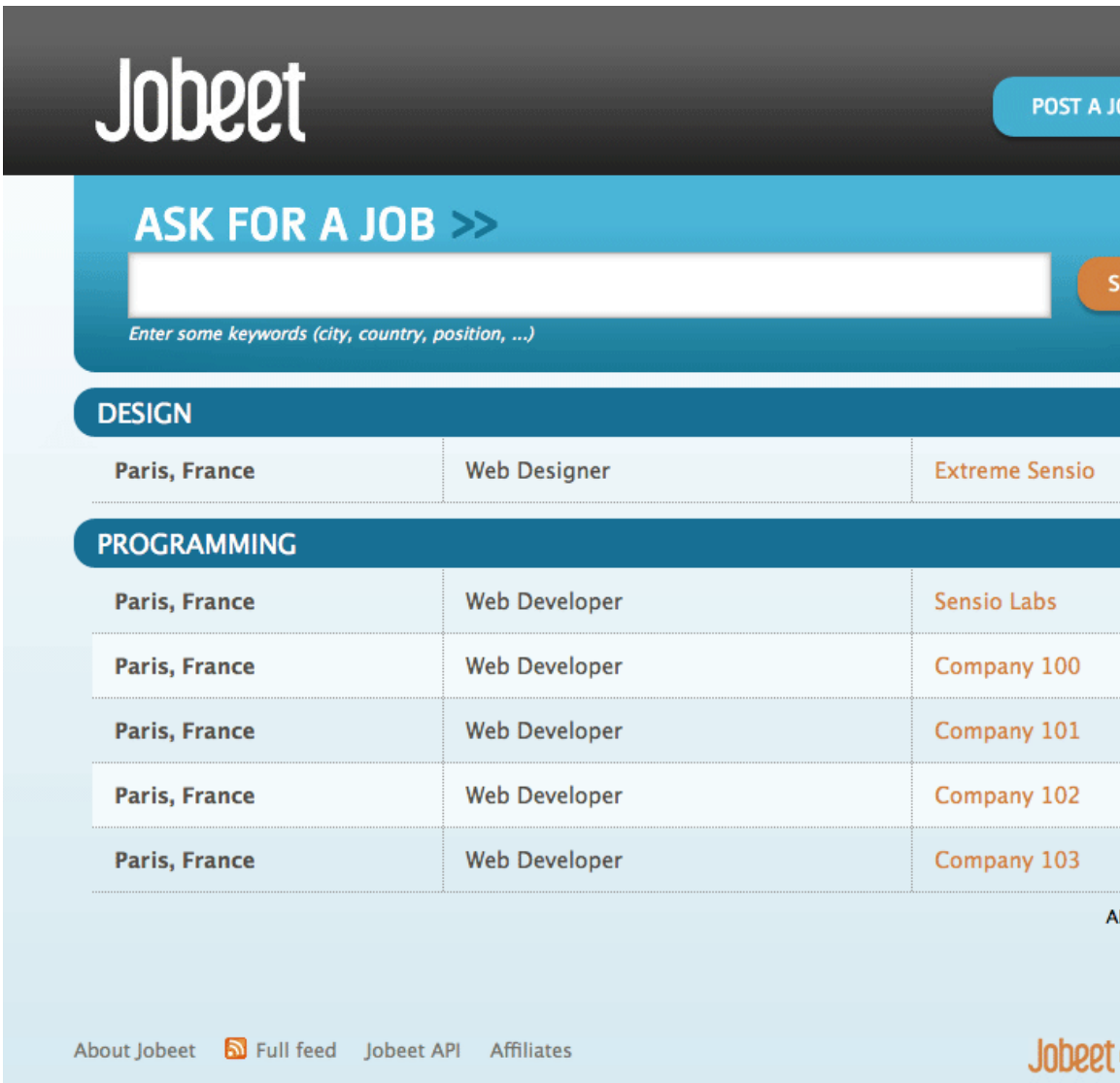
Comme vous pouvez le constater, nous avons refactorisé tout le code de `JobeetJobTable` afin d'utiliser la nouvelle méthode `addActiveJobsQuery()` qui est partagée. Notre code utilise maintenant la philosophie DRY (Don't Repeat Yourself).

La première fois qu'une partie de code est réutilisée, le copier peut être suffisant. Mais si vous en avez besoin pour d'autres utilisations, vous devez refactoriser toutes les méthodes faisant appel à la fonction ou méthode partagée comme nous venons de le faire.

Au lieu d'utiliser `execute()` puis de compter le nombre de résultats dans la méthode `countActiveJobs()`, nous avons utilisé la méthode `count()` qui est beaucoup plus rapide.

Nous venons de modifier beaucoup de fichiers pour cette simple fonctionnalité. Mais pour chaque ajout de code, nous avons essayé de le mettre dans la bonne couche de l'application et

nous avons également essayé de rendre le code réutilisable. Dans la foulée, nous avons refactorisé du code existant. C'est une méthode de travail typique quand vous travaillez sur un projet symfony. Dans la capture d'écran suivante, nous montrons que 5 emplois pour diminuer l'affichage, vous devriez en voir 10 (le paramètre `max_jobs_on_homepage`):



Création du module de la catégorie d'un emploi

Il est temps de créer le module `category` :

```
$ php symfony generate:module frontend category
```

Si vous avez créé le module, vous avez sûrement utilisé `doctrine:generate-module`. C'est très bien, mais comme nous n'aurons pas besoin de 90% du code généré, j'ai utilisé le `generate:module` qui crée un module vide.

Pourquoi ne pas ajouter une action `category` pour le module `job` ? Nous pourrions, mais comme le sujet principal de la page catégorie est une catégorie, il parait plus naturel de créer un module de `category` dédiée.

Lorsque vous accédez à la page catégorie, la route `category` devra trouver la catégorie associée avec la variable `slug`. Mais étant donné que slug n'est pas stocké dans la base de donnée et parce qu'il est impossible d'en déduire la catégorie, nous n'avons aucun moyen de trouver la catégorie pour le moment.

Mise à jour de la base de donnée

Nous devons ajouter la colonne `slug` à la table `category` :

Cette colonne slug peut être prise en charge par un comportement de Doctrine nommé Sluggable. Nous avons simplement besoin d'activer le comportement de notre modèle JobeetCategory et il prendra soin de tout pour vous.

```
# config/doctrine/schema.yml
JobeetCategory:
  actAs:
    Timestampable: ~
    Sluggable:
      fields: [name]
  columns:
    name:
      type: string(255)
      notnull:  true
```

A présent, slug est une colonne réelle et vous pouvez donc supprimer la méthode getSlug() du modèle JobeetCategory.

Le paramètre de la colonne slug est automatiquement pris en compte quand vous sauvegardez un enregistrement. La valeur de slug est construite à partir du champ name et est passée en objet.

Utilisez la tâche doctrine:build --all --and-load pour mettre à jour les tables de la base de donnée et recharger les données avec vos jeux de test :

```
$ php symfony doctrine:build --all --and-load --no-confirmation
```

Nous pouvons maintenant créer la méthode executeShow(). Remplacez le contenu du fichier des actions de category avec le code suivant :

```
// apps/frontend/modules/category/actions/actions.class.php
class categoryActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->category = $this->getRoute()->getObject();
  }
}
```

Puisque nous venons de supprimer la méthode executeIndex(), vous pouvez aussi supprimer le Template indexSuccess.php qui a été généré automatiquement (apps/frontend/modules/category/templates/indexSuccess.php).

Pour la dernière étape, il reste à créer le template showSuccess.php :

```
// apps/frontend/modules/category/templates/showSuccess.php
<?php use_stylesheet('jobs.css') ?>

<?php slot('title', sprintf('Jobs in the %s category', $category->getName())) ?>

<div class="category">
  <div class="feed">
    <a href="">Feed</a>
  </div>
  <h1><?php echo $category ?></h1>
</div>

<table class="jobs">
  <?php foreach ($category->getActiveJobs() as $i => $job): ?>
    <tr class="<?php echo fmod($i, 2) ? 'even' : 'odd' ?>">
      <td class="location">
        <?php echo $job->getLocation() ?>
      </td>
      <td class="position">
        <?php echo link_to($job->getPosition(), 'job_show_user', $job) ?>
      </td>
      <td class="company">
        <?php echo $job->getCompany() ?>
      </td>
```

```
    </tr>
  <?php endforeach; ?>
</table>
```

Partials

Notez que nous avons copié et collé la balise `<table>` afin de créer une liste d'emploi depuis le Template job `indexSuccess.php`. C'est mauvais. Il est temps d'apprendre un nouveau tour. Lorsque vous avez besoin de réutiliser une partie d'un Template, vous devez créer un **partial**. Un partial est un extrait du code du Template qui peut être partagé entre plusieurs Templates. Un partial est juste un autre Template qui commence par un caractère de soulignement (`_`).

Créez le fichier `_list.php` :

```
// apps/frontend/modules/job/templates/_list.php
<table class="jobs">
  <?php foreach ($jobs as $i => $job): ?>
    <tr class="<?php echo fmod($i, 2) ? 'even' : 'odd' ?>">
      <td class="location">
        <?php echo $job->getLocation() ?>
      </td>
      <td class="position">
        <?php echo link_to($job->getPosition(), 'job_show_user', $job) ?>
      </td>
      <td class="company">
        <?php echo $job->getCompany() ?>
      </td>
    </tr>
  <?php endforeach; ?>
</table>
```

Vous pouvez inclure un partial en utilisant le helper `include_partial()` :

```
<?php include_partial('job/list', array('jobs' => $jobs)) ?>
```

Le premier argument du helper `include_partial()` est le nom du partial (nom du module, un slash `/`, et le nom du partial sans le caractère de soulignement (`_`). Le second argument est un tableau contenant les variables à passer dans le partial.

Pourquoi ne pas utiliser la méthode `include()` intégrée à PHP plutôt que le helper `include_partial()` ? La principale différence est le support du cache intégré au helper `include_partial()`.

Remplaçez le bloc de code HTML `<table>` dans les deux templates par un appel `include_partial()` :

```
// in apps/frontend/modules/job/templates/indexSuccess.php
<?php include_partial('job/list', array('jobs' => $category-
>getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')))) ?>

// in apps/frontend/modules/category/templates/showSuccess.php
<?php include_partial('job/list', array('jobs' => $category->getActiveJobs())) ?>
```

Pagination de la liste

Conditions du Jour 2 :

"La liste est paginée avec 20 emplois par page."

Pour paginer une liste d'objets Doctrine, symfony fournit une classe dédiée : `sfDoctrinePager`. Dans l'action `category`, au lieu de passer les objets job au template `showSuccess`, nous allons passer un paginateur :

```
// apps/frontend/modules/category/actions/actions.class.php
public function executeShow(sfWebRequest $request)
{
  $this->category = $this->getRoute()->getObject();

  $this->pager = new sfDoctrinePager(
    'JobeetJob',
    sfConfig::get('app_max_jobs_on_category')
  );
```

```
  $this->pager->setQuery($this->category->getActiveJobsQuery());
  $this->pager->setPage($request->getParameter('page', 1));
  $this->pager->init();
}
```

La méthode `sfRequest::getParameter()` prend une valeur par défaut pour le second argument. Dans l'action ci-dessus, si le paramètre requis page n'existe pas, alors `getParameter()` retournera 1.

Le constructeur `sfDoctrinePager` utilise la classe du modèle et le nombre maximum d'items à afficher par page. Ajouter la dernière ligne au fichier de configuration :

```
# apps/frontend/config/app.yml
all:
  active_days:          30
  max_jobs_on_homepage: 10
  max_jobs_on_category: 20
```

La méthode `sfDoctrinePager::setQuery()` prend un objet `Doctrine_Query` à utiliser lors de la sélection des items de la base de données.

Ajoutez la méthode `getActiveJobsQuery()` :

```
// lib/model/doctrine/JobeetCategory.class.php
public function getActiveJobsQuery()
{
  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.category_id = ?', $this->getId());

  return Doctrine_Core::getTable('JobeetJob')->addActiveJobsQuery($q);
}
```

Maintenant que nous avons défini la méthode `getActiveJobsQuery()`, nous pouvons refactoriser les autres méthodes du modèle `JobeetCategory` qui l'utilise : [php] // lib/model/doctrine/JobeetCategory.class.php public function getActiveJobs($max = 10) { $q = $this->getActiveJobsQuery() ->limit($max);

```
  return $q->execute();
}

public function countActiveJobs()
{
  return $this->getActiveJobsQuery()->count();
}
```

Et pour finir, mettons le Template à jour :

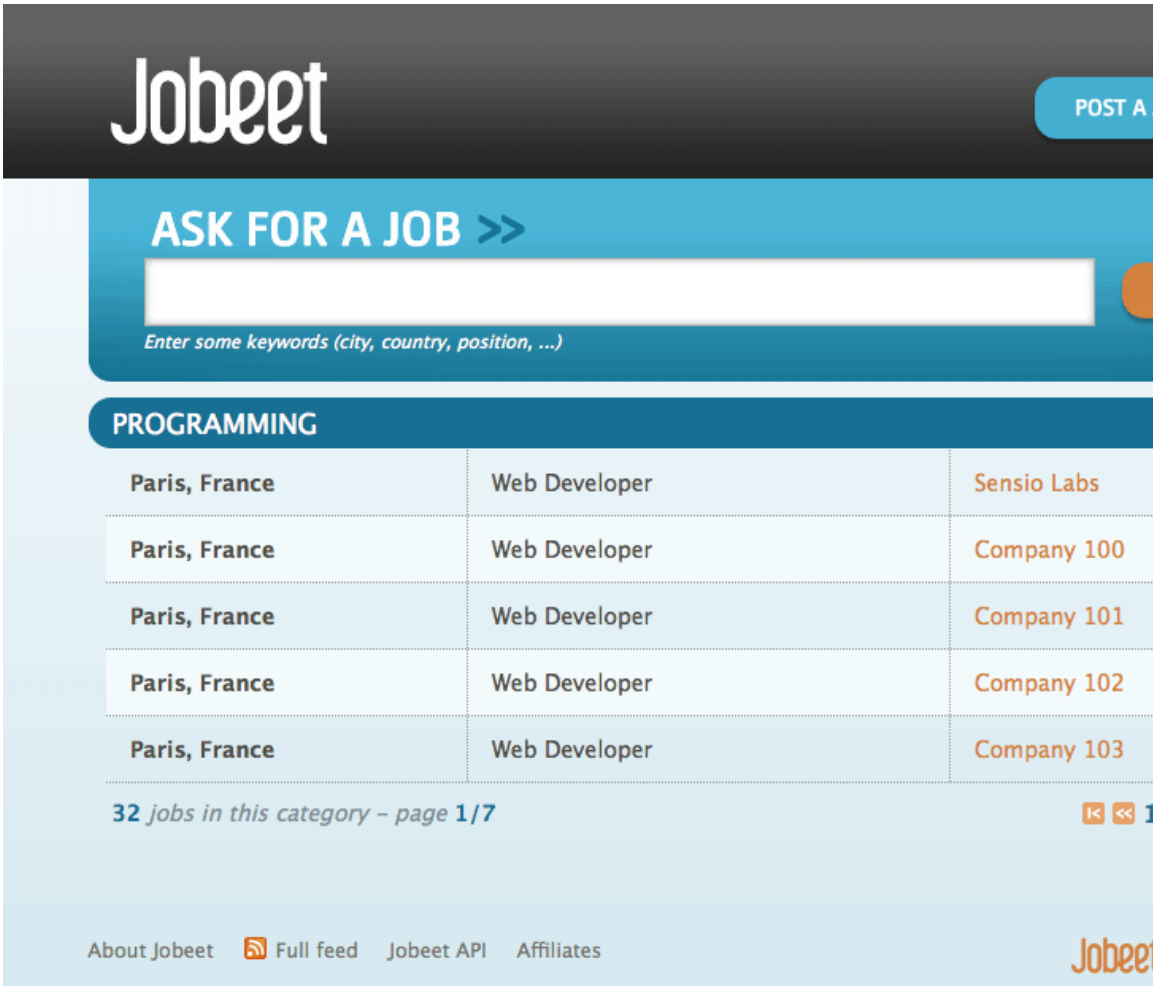
```
<!-- apps/frontend/modules/category/templates/showSuccess.php -->
<?php use_stylesheet('jobs.css') ?>

<?php slot('title', sprintf('Jobs in the %s category', $category->getName())) ?>

<div class="category">
  <div class="feed">
    <a href="">Feed</a>
  </div>
  <h1><?php echo $category ?></h1>
</div>

<?php include_partial('job/list', array('jobs' => $pager->getResults())) ?>

<?php if ($pager->haveToPaginate()): ?>
  <div class="pagination">
    <a href="<?php echo url_for('category', $category) ?>?page=1">
      <img src="/images/first.png" alt="First page" title="First page" />
    </a>

    <a href="<?php echo url_for('category', $category) ?>?page=<?php echo $pager-
>getPreviousPage() ?>">
      <img src="/images/previous.png" alt="Previous page" title="Previous page" />
```

```
    </a>

    <?php foreach ($pager->getLinks() as $page): ?>
      <?php if ($page == $pager->getPage()): ?>
        <?php echo $page ?>
      <?php else: ?>
        <a href="<?php echo url_for('category', $category) ?>?page=<?php echo $page ?>"><?php echo $page ?></a>
      <?php endif; ?>
    <?php endforeach; ?>

    <a href="<?php echo url_for('category', $category) ?>?page=<?php echo $pager->getNextPage() ?>">
      <img src="/images/next.png" alt="Next page" title="Next page" />
    </a>

    <a href="<?php echo url_for('category', $category) ?>?page=<?php echo $pager->getLastPage() ?>">
      <img src="/images/last.png" alt="Last page" title="Last page" />
    </a>
  </div>
<?php endif; ?>

<div class="pagination_desc">
  <strong><?php echo count($pager) ?></strong> jobs in this category

  <?php if ($pager->haveToPaginate()): ?>
    - page <strong><?php echo $pager->getPage() ?>/<?php echo $pager->getLastPage() ?></strong>
  <?php endif; ?>
</div>
```

L'essentiel de ce code traite des liens vers d'autres pages. Voici la liste des méthodes `sfDoctrinePager` utilisées dans ce Template :

- `getResults()` : Retourne un tableau d'objets Doctrine de la page actuelle
- `getNbResults()` : Retourne le nombre total de résultats
- `haveToPaginate()` : Retourne `true` s'il y a plus d'une page
- `getLinks()` : Retourne une liste de liens vers des pages à afficher
- `getPage()` : Retourne le numéro de la page actuelle
- `getPreviousPage()` : Retourne le numéro de la page précédente
- `getNextPage()` : Retourne le numéro de la page suivante
- `getLastPage()` : Retourne le numéro de la dernière page

Comme `sfDoctrinePager` implémente également les interfaces `Iterator` et `Countable`, vous pouvez utiliser la fonction `count()` pour obtenir le nombre de résultats au lieu de la méthode `getNbResults()`.



À demain

Si vous avez travaillé sur votre propre implémentation d'hier et le sentiment que vous n'avez pas appris beaucoup aujourd'hui, cela signifie que vous êtes habituer à la philosophie de symfony. Le processus pour ajouter une nouvelle fonctionnalité à un site web symfony est toujours le même : réfléchir sur les URL, créer certaines actions, mettre à jour le modèle et écrire quelques Templates. Et, si vous pouvez appliquer certaines bonnes pratiques de développement en plus, vous deviendrez un maître symfony très rapidement.

Demain sera le début d'une nouvelle semaine pour Jobeet. Pour fêter ça, nous allons parler d'un tout nouveau sujet : les tests.

« Jour 6 : Aller plus loin avec le Modèle

Jour 8 : Les tests unitaires »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

development, training, consulting.
Sensio Labs also supports several large Open-Source projects.

Swift Mailer - Mailing library
Twig - Template library
Pirum - PEAR channel server

Books -
Confere
Confere

Practical symfony
Jour 8 : Les tests unitaires

Au cours des deux derniers jours, nous avons examiné tous les éléments appris au cours des cinq premiers chapitres du Pratical symfony book, afin de personnaliser les fonctionnalités de Jobeet et d'en ajouter de nouvelles. Dans le processus, nous avons également abordé d'autres fonctionnalités plus avancées de symfony.

Aujourd'hui, nous allons commencer à parler de quelque chose de complètement différent : les **tests** automatisés. Comme le sujet est assez vaste, il nous faudra deux jours complets pour tout couvrir.

Les tests dans symfony

Il existe deux différents types de tests automatisés dans symfony: les **tests unitaires** et les **tests fonctionnels**.

Les tests unitaires vérifient que chaque méthode et chaque fonction fonctionne correctement. Chaque test doit être aussi indépendante que possible des autres.

D'autre part, des tests fonctionnels vérifient que l'application résultante se comporte correctement dans son ensemble.

Tous les tests dans symfony sont situés sous le répertoire `test/` du projet. Il contient deux sous-répertoires, un pour les tests unitaires (`test/unit/`) et un pour les tests fonctionnels (`test/functional/`).

Les tests unitaires seront couverts dans ce chapitre et le suivant sera consacré aux tests fonctionnels.

Les tests unitaires

L'écriture des tests unitaires est peut-être l'une des pratiques les plus difficiles du développement web à mettre en place. Car les développeurs web ne sont pas vraiment utilisé pour tester leur travail, beaucoup de questions se posent : Dois-je écrire des tests avant d'implémenter une fonctionnalité ? Que dois-je tester ? Mes tests doivent couvrir chaque cas limite ? Comment puis-je être sûr que tout est bien testé ? Mais d'habitude, la première question est beaucoup plus fondamentale : où commencer?

Même si nous insistons fortement sur les tests, l'approche de symfony est pragmatique : il est toujours préférable d'avoir quelques tests que pas de test du tout. Avez-vous déjà beaucoup de code sans aucun test ? Pas de problème. Vous n'avez pas besoin d'avoir une suite de tests complète pour bénéficier des avantages des tests. Commencez par l'ajout de test lorsque vous trouvez un bogue dans votre code. Au fil du temps, votre code va devenir de meilleure qualité, la couverture du code va augmenter, et vous deviendrez plus confiant sur ce sujet. En commençant avec une approche pragmatique, vous vous sentirez plus à l'aise avec les tests dans le temps. L'étape suivante consiste à écrire des tests pour des nouvelles fonctionnalités. En peu temps, vous allez devenir accro aux tests.

Le problème avec la plupart des bibliothèques de test est leur difficulté d'apprentissage. C'est pourquoi Symfony fournit une bibliothèque très simple de test, **lime**, pour faire de l'écriture de test avec une incroyable facilité.

Même si ce tutoriel décrit intensivement la bibliothèque intégrée lime, vous pouvez employer n'importe quelle bibliothèque de test, comme l'excellente bibliothèque PHPUnit.

Le framework de test *lime*

Tous les tests unitaires écrits avec le framework lime débutent avec le même code :

```
require_once dirname(__FILE__).'/../bootstrap/unit.php';

$t = new lime_test(1);
```

Tout d'abord, le fichier d'amorçage `unit.php` est inclus pour initialiser un certain nombre de choses. Puis, un nouvel objet `lime_test` est créé et le nombre de tests prévus à l'exécution est passé comme un argument.

Le plan permet à lime d'afficher un message d'erreur au cas où trop peu de tests seraient exécutés (par exemple quand un test génère une erreur fatale PHP).

Les tests fonctionnent en appelant une méthode ou une fonction avec un ensemble d'entrées prédéfinies, puis en comparant les résultats avec ceux escomptés. Cette comparaison permet de déterminer si un test réussit ou échoue.

Pour faciliter la comparaison, l'objet `lime_test` fournit plusieurs méthodes :

Méthode	Description
`ok($test)`	Teste une condition et passe si elle est vrai
`is($value1, $value2)`	Compare deux valeurs et passe si elles sont égales (==)
`isnt($value1, $value2)`	Compare deux valeurs et passe si elles ne sont pas égales
`like($string, $regexp)`	Teste une chaîne à une expression régulière
`unlike($string, $regexp)`	Vérifie qu'une chaîne ne correspond pas à une expression régulière
`is_deeply($array1, $array2)`	Vérifie que les deux tableaux ont les mêmes valeurs

Vous pouvez vous demander pourquoi lime définit tant de méthodes de test, car tous les tests peuvent être écrits juste en employant la méthode `ok()`. L'avantage du choix des méthodes se situe dans des messages d'erreur beaucoup plus explicites en cas de test échoué et pour une lisibilité améliorée des tests.

L'objet `lime_test` fournit également d'autres méthodes de test pratique :

Méthode	Description
`fail()`	Echoue toujours -- Utile pour tester les exceptions
`pass()`	Passe toujours -- Utile pour tester les exceptions
`skip($msg, $nb_tests)`	Compte pour $nb_tests - utile pour les tests conditionnels
`todo()`	Compte pour un test - utile pour les tests qui ne sont pas encore écrits

Enfin, la méthode `comment($msg)` renvoie un commentaire, mais n'exécute aucun test.

Exécution des tests unitaires

Tous les tests unitaires sont stockés dans le répertoire `test/unit/`. Par convention, les tests sont nommés d'après la classe qu'ils testent et suffixé par `Test`. Même si vous pouvez organiser les fichiers sous le répertoire `test/unit/` comme vous le désirez, nous vous conseillons de reproduire la structure du répertoire `lib/`.

Pour illustrer le test unitaire, nous allons tester la classe `Jobeet`.

Créez un fichier `test/unit/JobeetTest.php` et copiez le code suivant à l'intérieur :

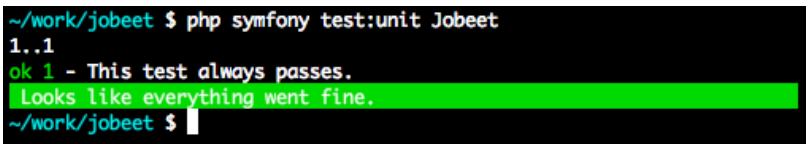
```
// test/unit/JobeetTest.php
require_once dirname(__FILE__).'/../bootstrap/unit.php';

$t = new lime_test(1);
$t->pass('This test always passes.');
```

Pour lancer les tests, vous pouvez exécuter le fichier directement :

```
$ php test/unit/JobeetTest.php
```

Ou utilisez la tâche `test:unit` :

```
$ php symfony test:unit Jobeet
```



Windows en ligne de commande ne peut malheureusement pas mettre en évidence les résultats des tests en couleur rouge ou verte. Mais si vous utilisez Cygwin, vous pouvez forcer symfony pour utiliser des couleurs en passant l'option `--color` à la tâche.

Tester *slugify*

Commençons notre voyage dans le monde merveilleux des tests unitaires en écrivant des tests pour la méthode `Jobeet::slugify()`.

Nous avons créé la méthode `~slug|Slug~ify()` durant la journée 5 pour nettoyer une chaîne de sorte qu'elle ne puisse pas être dangereuse dans une URL. La conversion consiste à certaines transformations de base comme la conversion de tous les caractères non-ASCII par un tiret (`-`) ou de convertir la chaîne en minuscules:

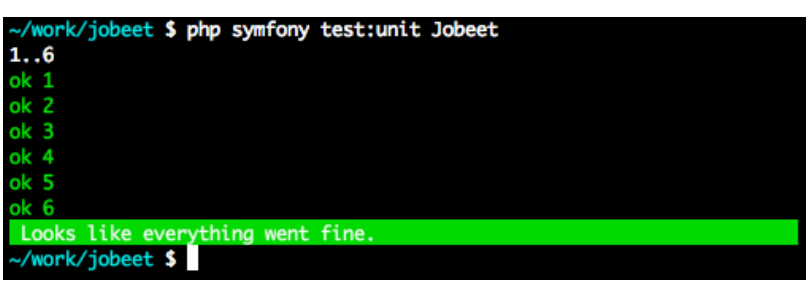
Entrée	Sortie
Sensio Labs	sensio-labs
Paris, France	paris-france

Remplacez le contenu du fichier de test avec le code suivant:

```
// test/unit/JobeetTest.php
require_once dirname(__FILE__).'/../bootstrap/unit.php';

$t = new lime_test(6);

$t->is(Jobeet::slugify('Sensio'), 'sensio');
$t->is(Jobeet::slugify('sensio labs'), 'sensio-labs');
$t->is(Jobeet::slugify('sensio   labs'), 'sensio-labs');
$t->is(Jobeet::slugify('paris,france'), 'paris-france');
$t->is(Jobeet::slugify('  sensio'), 'sensio');
$t->is(Jobeet::slugify('sensio  '), 'sensio');
```

Si vous jetez un coup d'œil aux tests que nous avons écrit, vous remarquerez que chaque ligne teste qu'une seule chose. C'est quelque chose que vous devez garder à l'esprit lors de l'écriture des tests unitaires. Testez une chose à la fois.

Vous pouvez maintenant exécuter le fichier de test. Si tous les tests sont réussis, comme on peut s'y attendre, vous pourrez profiter de la "*barre verte*". Sinon, la fameuse "*barre rouge*" vous avertira que certains tests ne passent pas et que vous avez besoin de les corriger.



Si un test échoue, l'affichage sera de vous donner quelques informations sur la raison de cet échec, mais si vous avez des centaines de tests dans un fichier, il peut être difficile d'identifier rapidement le problème qui échoue.

Toutes les méthodes de test de lime prennent une chaîne en dernier argument qui sert pour la description du test. C'est très pratique, car elle vous oblige à décrire ce que font vraiment les tests. Elle peut aussi servir comme une forme de documentation du comportement attendu d'une méthode. Ajoutons quelques messages au fichier de test `slugify`:

```php
require_once dirname(__FILE__).'/../bootstrap/unit.php';

$t = new lime_test(6);

$t->comment('::slugify()');
$t->is(Jobeet::slugify('Sensio'), 'sensio', '::slugify() converts all characters to lower case');
$t->is(Jobeet::slugify('sensio labs'), 'sensio-labs', '::slugify() replaces a white space by a -');
$t->is(Jobeet::slugify('sensio   labs'), 'sensio-labs', '::slugify() replaces several white spaces by a single -');
$t->is(Jobeet::slugify('  sensio'), 'sensio', '::slugify() removes - at the beginning of a string');
$t->is(Jobeet::slugify('sensio  '), 'sensio', '::slugify() removes - at the end of a string');
$t->is(Jobeet::slugify('paris,france'), 'paris-france', '::slugify() replaces non-ASCII characters by a -');
```

```
~/work/jobeet $ php symfony test:unit Jobeet
1..6
# ::slugify()
ok 1 - ::slugify() converts all characters to lower case
ok 2 - ::slugify() replaces a white space by a -
ok 3 - ::slugify() replaces several white spaces by a single -
ok 4 - ::slugify() replaces non-ASCII characters by a -
ok 5 - ::slugify() removes - at the beginning of a string
ok 6 - ::slugify() removes - at the end of a string
Looks like everything went fine.
~/work/jobeet $
```

La chaîne de description du test est également un outil précieux pour comprendre ce test lors d'un essai. Vous pouvez voir une structure dans les chaînes de test : c'est des phrases décrivant comment la méthode doit se comporter et elles commencent toujours avec le nom de la méthode à tester.

Couverture de code

Lorsque vous écrivez des tests, il est facile d'oublier une partie du code.

Pour vous aider à vérifier que tout votre code est bien testé, symfony fournit la tâche `test:coverage`. Passez cette tâche à un fichier ou à un répertoire de test et un fichier ou un répertoire lib comme arguments et il vous indiquera la couverture de votre code :

```
$ php symfony test:coverage test/unit/JobeetTest.php lib/Jobeet.class.php
```

Si vous voulez savoir quelles lignes ne sont pas couverts par vos tests, passer l'option `--detailed` :

```
$ php symfony test:coverage --detailed test/unit/JobeetTest.php
lib/Jobeet.class.php
```

Gardez à l'esprit que lorsque la tâche indique que votre code est entièrement testé unitairement, cela signifie juste que chaque ligne a été exécuté, mais cela ne signifie pas que tous les cas limites ont été testés.

Comme `test:coverage` repose sur XDebug pour collecter ses informations, vous devez l'installer et l'activer en premier.

Ajout de tests pour les nouvelles fonctionnalités

Le slug pour une chaîne vide est une chaîne vide. Vous pouvez le tester, il va fonctionner. Mais une chaîne vide dans une URL, ce n'est pas une bonne idée. Modifions donc la méthode `slugify()` de sorte qu'elle retourne la chaîne "n-a" dans le cas d'une chaîne vide.

Vous pouvez écrire le premier test, puis mettre à jour la méthode, ou l'inverse. C'est vraiment une question de goût mais l'écriture du test vous donne d'abord la confiance que votre code implémente réellement ce que vous avez prévu :

```php
$t->is(Jobeet::slugify(''), 'n-a', '::slugify() converts the empty string to n-a');
```

Cette méthodologie de développement, dans laquelle vous écrivez d'abord les tests puis

l'implémentation des fonctionnalités, est appelé Test Driven Development (TDD).

Si vous lancez les tests maintenant, vous devez avoir une barre rouge. Sinon, cela signifie que la fonctionnalité est déjà implémenté ou que votre test ne teste pas ce qu'il est censé tester.

Maintenant, modifiez la classe Jobeet et ajoutez la condition suivante au début :

```
// lib/Jobeet.class.php
static public function slugify($text)
{
  if (empty($text))
  {
    return 'n-a';
  }

  // ...
}
```

Le test doit maintenant passer comme prévu et vous pouvez profiter de la barre verte. Mais seulement si vous vous êtes rappelés de mettre à jour le plan de test. Sinon, vous aurez un message indiquant que vous avez prévu six tests et vous en avez exécuté un de plus. Avoir le nombre de tests prévus est important, car il vous tiendra au courant dès le début si le script de test échoue.

Ajout de test en raison d'un bug

Disons que le temps a passé et l'un de vos utilisateurs vous rapporte un bogue bizarre: certains liens des emplois pointent vers une page d'erreur 404. Après quelques recherches, vous trouvez pour une raison quelconque, que ces emplois ont une société, une position ou un emplacement vide.

Comment cela est-il possible?

Vous regardez à travers les enregistrements de la base de données et les colonnes ne sont certainement pas vide. Vous réfléchissez pendant un moment, et hop, vous trouvez la cause. Lorsqu'une chaîne ne contient que des caractères non-ASCII, la méthode `slugify()` la transforme en une chaîne vide. Tellement heureux d'avoir trouvé la cause, vous ouvrez la classe Jobeet et vous corrigez le problème immédiatement. C'est une mauvaise idée. Premièrement, nous allons ajouter un test :

```
$t->is(Jobeet::slugify(' - '), 'n-a', '::slugify() converts a string that only
contains non-ASCII characters to n-a');
```

```
~/work/jobeet $ php symfony test:unit Jobeet
1..8
# ::slugify()
ok 1 - ::slugify() converts all characters to lower case
ok 2 - ::slugify() replaces a white space by a -
ok 3 - ::slugify() replaces several white spaces by a single -
ok 4 - ::slugify() replaces non-ASCII characters by a -
ok 5 - ::slugify() removes - at the beginning of a string
ok 6 - ::slugify() removes - at the end of a string
ok 7 - ::slugify() replaces the empty string by n-a
not ok 8 - ::slugify() replaces a string that only contains non-ASCII ch
#     Failed test (/Users/fabien/work/symfony/dev/1.2/lib/vendor/lime/li
#           got: ''
#      expected: 'n-a'
 Looks like you failed 1 tests of 8.
~/work/jobeet $
```

Après avoir vérifié que le test ne passe pas, modifiez la classe Jobeet et passez la vérification de la chaîne vide à la fin de la méthode :

```
static public function slugify($text)
{
  // ...

  if (empty($text))
  {
    return 'n-a';
  }

  return $text;
}
```

Le nouveau test passe désormais, comme tous les autres. Le `slugify()` avait un bug en dépit de notre couverture à 100%.

Vous ne pouvez pas penser à tous les cas limites lors de l'écriture des tests et c'est très bien. Mais quand vous en découvrez un, vous devez écrire un test avant de corriger votre code. Cela signifie également que votre code va s'améliorer au fil du temps, ce qui est toujours une bonne chose.

Vers une meilleure méthode *slugify*

Vous savez probablement que symfony a été créé par des Français, nous allons donc ajouter un test avec un mot français qui contient un «accent» :

```
$t->is(Jobeet::slugify('Développeur Web'), 'developpeur-web', '::slugify()
removes accents');
```

Le test doit échouer. Au lieu de remplacer é par e, la méthode `slugify()` l'a remplacé par un tiret (`-`). C'est un problème difficile, appelé *translittération*. En espérant que vous avez installé "iconviconv", il fera le travail pour nous. Remplacez le code de la méthode `slugify` avec le texte suivant :

```
// code derived from http://php.vrana.cz/vytvoreni-pratelskeho-url.php
static public function slugify($text)
{
  // replace non letter or digits by -
  $text = preg_replace('#[^\\pL\d]+#u', '-', $text);

  // trim
  $text = trim($text, '-');

  // transliterate
  if (function_exists('iconv'))
  {
    $text = iconv('utf-8', 'us-ascii//TRANSLIT', $text);
  }

  // lowercase
  $text = strtolower($text);

  // remove unwanted characters
  $text = preg_replace('#[^-\w]+#', '', $text);

  if (empty($text))
  {
    return 'n-a';
  }

  return $text;
}
```

N'oubliez pas de sauvegarder tous vos fichiers PHP avec l'encodage UTF-8, car cela est l'encodage de symfony par défaut et celle utilisée par «iconv» pour faire la translitération.

Également modifier le fichier de test pour exécuter le test que si "iconv" est disponible :

```
if (function_exists('iconv'))
{
  $t->is(Jobeet::slugify('Développeur Web'), 'developpeur-web', '::slugify()
removes accents');
}
else
{
  $t->skip('::slugify() removes accents - iconv not installed');
}
```

Tests unitaires de Doctrine

Configuration de la base de données

Les tests unitaires d'une classe modèle Doctrine est un peu plus complexe, car elle requiert une connexion à la base de données. Vous avez déjà celle que vous utilisez pour votre

développement, mais c'est une bonne habitude de créer une base de données dédiée pour des tests.

Durant le jour 1, nous avons introduit les environnements comme un moyen pour faire varier les paramètres d'une application. Par défaut, tous les tests de symfony sont lancées dans l'environnement de `test`, donc nous allons configurer une base de données différentes pour l'environnement de `test` :

```
$ php symfony configure:database --name=doctrine --class=sfDoctrineDatabase --env=test
"mysql:host=localhost;dbname=jobeet_test" root mYsEcret
```

L'option env dit à la tâche que la configuration de la base de données est seulement pour l'environnement de `test`. Lorsque nous avons utilisé cette tâche pendant le jour 3, nous n'avions passé aucune option env, car la configuration a été appliquée à tous les environnements.

Si vous êtes curieux, ouvrez le fichier de configuration `config/databases.yml` pour voir comment symfony fait. Il est facile de modifier la configuration en fonction de l'environnement.

Maintenant que nous avons configuré la base de données, nous pouvons l'amorcer en employant la tâche `doctrine:insert-sql` :

```
$ mysqladmin -uroot -pmYsEcret create jobeet_test
$ php symfony doctrine:insert-sql --env=test
```

Principes de configuration dans symfony

Au cours du jour 4, nous avons vu que les paramètres provenants des fichiers de configuration peuvent être définis à différents niveaux.

Ces paramètres peuvent également être dépendant de l'environnement. Cela est vrai pour la plupart des fichiers de configuration que nous avons utilisé jusqu'à présent : `databases.yml`, `app.yml`, `view.yml`, et `settings.yml`. Dans tous ces fichiers, la clé principale est l'environnement, la clé `all` indiquant que ses paramètres sont sur tous les environnements :

```
# config/databases.yml
dev:
  doctrine:
    class: sfDoctrineDatabase

test:
  doctrine:
    class: sfDoctrineDatabase
    param:
      dsn: 'mysql:host=localhost;dbname=jobeet_test'

all:
  doctrine:
    class: sfDoctrineDatabase
    param:
      dsn: 'mysql:host=localhost;dbname=jobeet'
      username: root
      password: null
```

Données de test

Maintenant que nous avons une base de données dédiée pour nos tests, nous avons besoin d'un moyen pour charger les données de test. Durant le chapitre 3, vous avez appris à utiliser la tâche `doctrine:data-load`, mais pour des tests, nous avons besoin de recharger les données chaque fois que nous les exécutons pour mettre la base de données dans un état connu.

La tâche `doctrine:data-load` utilise en interne la méthode `Doctrine_Core::loadData()` pour charger les données :

```
Doctrine_Core::loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

L'objet `sfConfig` peut être utilisé pour obtenir le chemin complet d'un sous-répertoire du

projet. Son utilisation permet à la structure de répertoire par défaut pour être personnalisée.

La méthode `loadData()` prend un répertoire ou un fichier comme premier argument. Elle peut également prendre un tableau de répertoires et/ou de fichiers.

Nous avons déjà créé quelques données initiales dans le répertoire `data/fixtures/`. Pour les tests, nous mettrons les jeux de test dans le répertoire `test/fixtures/`. Ces jeux de test seront utilisés pour les tests unitaires et fonctionnels de Doctrine.

Pour l'instant, copiez les fichiers à partir `data/fixtures/` vers le répertoire `test/fixtures/`.

Testing *JobeetJob*

Nous allons créer quelques tests unitaires pour la classe du modèle `JobeetJob`.

Comme tous nos tests unitaires pour Doctrine débuteront avec le même code, créez un fichier `Doctrine.php` dans le répertoire de test `bootstrap/` avec le code suivant :

```
// test/bootstrap/Doctrine.php
include(dirname(__FILE__).'/unit.php');

$configuration = ProjectConfiguration::getApplicationConfiguration( 'frontend',
'test', true);

new sfDatabaseManager($configuration);

Doctrine_Core::loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

Le script est assez explicite :

- En ce qui concerne les contrôleurs frontaux, on initialise un objet de configuration pour l'environnement `test` :

  ```
  $configuration = ProjectConfiguration::getApplicationConfiguration( 'frontend',
  'test', true);
  ```

- Nous créons un gestionnaire de base de données. Il initialise la connexion Doctrine en chargeant le fichier de configuration `databases.yml`.

  ```
  new sfDatabaseManager($configuration);
  ```

- Nous chargeons nos données de test en utilisant `Doctrine_Core::loadData()`:

  ```
  Doctrine_Core::loadData(sfConfig::get('sf_test_dir').'/fixtures');
  ```

Doctrine ne se connecte à la base de données que si elle a des instructions SQL à exécuter.

Maintenant que tout est en place, nous pouvons commencer à tester la classe `JobeetJob`.

Premièrement, nous avons besoin de créer le fichier `JobeetJobTest.php` dans `test/unit/model`:

```
// test/unit/model/JobeetJobTest.php
include(dirname(__FILE__).'/../../bootstrap/Doctrine.php');

$t = new lime_test(1);
```

Puis, nous allons commencer par ajouter un test pour la méthode `getCompanySlug()` :

```
$t->comment('->getCompanySlug()');
$job = Doctrine_Core::getTable('JobeetJob')->createQuery()->fetchOne();
$t->is($job->getCompanySlug(), Jobeet::slugify($job->getCompany()), '-
>getCompanySlug() return the slug for the company');
```

Remarquez que nous ne testons que la méthode `getCompanySlug()` et pas si le slug est correct ou non, car nous l'avons déjà testé ailleurs.

L'écriture des tests pour la méthode `save()` est légèrement plus complexe :

```
$t->comment('->save()');
$job = create_job();
$job->save();
$expiresAt = date('Y-m-d', time() + 86400 * sfConfig::get('app_active_days'));
$t->is($job->getDateTimeObject('expires_at')->format('Y-m-d'), $expiresAt, '->save()
```

```
updates expires_at if not set');

$job = create_job(array('expires_at' => '2008-08-08'));
$job->save();
$t->is($job->getDateTimeObject('expires_at')->format('Y-m-d'), '2008-08-08', '->save()
does not update expires_at if set');

function create_job($defaults = array())
{
  static $category = null;

  if (is_null($category))
  {
    $category = Doctrine_Core::getTable('JobeetCategory')
      ->createQuery()
      ->limit(1)
      ->fetchOne();
  }

  $job = new JobeetJob();
  $job->fromArray(array_merge(array(
    'category_id'  => $category->getId(),
    'company'      => 'Sensio Labs',
    'position'     => 'Senior Tester',
    'location'     => 'Paris, France',
    'description'  => 'Testing is fun',
    'how_to_apply' => 'Send e-Mail',
    'email'        => 'job@example.com',
    'token'        => rand(1111, 9999),
    'is_activated' => true,
  ), $defaults));

  return $job;
}
```

Chaque fois que vous ajoutez des tests, n'oubliez pas de mettre à jour le nombre de tests prévus (le plan) dans la méthode du constructeur `lime_test`. Pour le fichier `JobeetJobTest`, vous devez le changer de 1 à 3.

Test sur les autres classes Doctrine

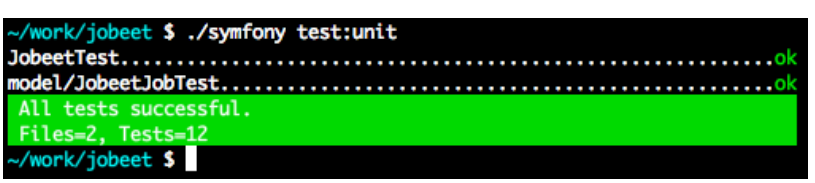
Vous pouvez maintenant ajouter des tests pour toutes les autres classes Doctrine. Comme vous êtes habitués maintenant à utiliser le processus d'écriture des tests unitaires, cela devrait être assez facile.

Validation des tests unitaires

La tâche `test:unit` peut également être utilisé pour lancer tous les tests unitaires pour un projet :

```
$ php symfony test:unit
```

La tâche affiche pour chaque fichier test s'il passe ou s'il échoue :



Si la tâche `test:unit` retourne un "état douteux" pour un fichier, il indique quel script échoue avant la fin. L'exécution du fichier de test à lui seul vous donne le message d'erreur exact.

À demain

Même si les tests d'une application sont assez importants, je sais que certains d'entre vous aurez pu être tenté de sauter le tutoriel d'aujourd'hui. Je suis content que vous ne l'ayez pas fait.

Pour sûr, la compréhension de symfony est l'étude de toutes les grandes fonctionnalités du framework fournit, mais c'est aussi sa philosophie de développement et les meilleures pratiques qu'il préconise. Et le test est un d'entre eux. Tôt ou tard, les tests unitaires économiseront des jours pour vous. Ils vous donnent une confiance solide de votre code et la liberté de le refactoriser sans crainte. Les tests unitaires sont une protection qui vous alertera si vous cassez quelque chose. Le framework symfony lui-même a plus de 9000 tests.

Demain, nous allons écrire quelques tests fonctionnels pour les modules `job` et `category`. Jusque-là, prenez le temps d'écrire plus de tests unitaires pour les classes du modèle Jobeet.

« Jour 7 : Jouons avec la page Catégorie

Jour 9 : Les tests fonctionnels »

Questions & Feedback

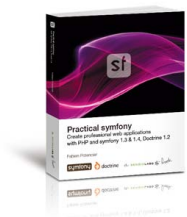
If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 9 : Les tests fonctionnels

Dans le chapitre précédent, nous avons vu comment faire des tests unitaires sur nos classes Jobeet en utilisant la bibliothèque intégrée lime avec symfony.

Dans ce chapitre, nous allons écrire des tests fonctionnels pour les fonctionnalités que nous avons déjà mis en œuvre dans les modules `job` et `category`.



Les tests fonctionnels

Les tests fonctionnels sont un excellent outil pour tester votre application de bout en bout : de la requête faite par un navigateur jusqu'à la réponse envoyée par le serveur. Ils testent toutes les couches d'une application : le routage, le modèle, les actions et les Templates. Ils sont très similaires à ce que vous avez sans doute déjà fait manuellement : chaque fois que vous ajoutez ou modifiez une action, vous devez aller dans le navigateur et vérifier que tout fonctionne comme prévu en cliquant sur les liens et en vérifiant les éléments sur la page rendue. En d'autres termes, vous exécutez un scénario correspondant au cas d'utilisation que vous venez de mettre en œuvre.

Comme le processus est manuel, il est complexe et assujetti aux erreurs. Chaque fois que vous changer quelque chose dans votre code, vous devez faire défiler tous les scénarios pour vérifier que vous n'avez rien cassé. C'est insensé. Les tests fonctionnels dans symfony fournissent un moyen de décrire facilement des scénarios. Chaque scénario peut alors être automatiquement lu maintes et maintes fois en simulant l'expérience d'un utilisateur dans un navigateur. Comme les tests unitaires, ils vous donnent la confiance pour coder en paix.

Les tests fonctionnels du framework ne remplace pas les outils tels que "Selenium". Selenium s'exécute directement dans le navigateur pour automatiser les tests sur plusieurs plateformes et navigateurs. Et en tant que tel, il est en mesure de tester le JavaScript de votre application.

La classe *sfBrowser*

Dans Symfony, les tests fonctionnels sont gérés via un navigateur spécial, implémenté par la classe `sfBrowser`. Elle agit comme un navigateur taillé sur mesure pour votre application et elle est directement connectée à lui, sans la nécessité d'un serveur web. Elle vous donne accès à tous les objets de symfony avant et après chaque requête, vous donnant la possibilité de les introspecter et d'effectuer les vérifications que vous voulez par programmation.

`sfBrowser` fournit des méthodes qui simule la navigation effectuée dans un navigateur classique :

Méthode	Description
`get()`	Obtient une URL
`post()`	Poste à une URL
`call()`	Appelle une URL (utilisé pour les méthodes PUT et DELETE)
`back()`	Revient en arrière d'une page dans l'historique
`forward()`	Avance d'une page dans l'historique
`reload()`	Recharge la page actuelle
`click()`	Clique sur un lien ou sur un bouton
`select()`	Sélectionne un radiobutton ou un checkbox
`deselect()`	Dé-sélectionne un radiobutton ou un checkbox

`restart()`	Redémarre le navigateur

Voici quelques exemples d'utilisation des méthodes `sfBrowser` :

```
$browser = new sfBrowser();

$browser->
  get('/')->
  click('Design')->
  get('/category/programming?page=2')->
  get('/category/programming', array('page' => 2))->
  post('search', array('keywords' => 'php'))
;
```

`sfBrowser` contient des méthodes supplémentaires pour configurer le comportement du navigateur :

Méthode	Description
`setHttpHeader()`	Définit une entête HTTP
`setAuth()`	Définit les informations d'authentification de base
`setCookie()`	Définit un cookie
`removeCookie()`	Enlève un cookie
`clearCookies()`	Vide tous les cookies courants
`followRedirect()`	Suit une redirection

La classe *sfTestFunctional*

Nous disposons d'un navigateur, mais nous avons besoin d'un moyen pour connaître les objets de symfony pour faire le test. Cela peut être fait avec lime et certaines méthodes de `sfBrowser` comme `getResponse()` et `getRequest()` mais symfony fournit une meilleure façon.

Les méthodes de test sont fournis par une autre classe, `sfTestFunctional` qui prend une instance `sfBrowser` dans son constructeur. La classe `sfTestFunctional` délègue les tests aux objets **testeur**. Plusieurs testeurs sont livrés avec symfony et vous pouvez également créer les vôtres.

Comme nous l'avons vu dans le chapitre 8, les tests fonctionnels sont stockés dans le répertoire `test/functional/`. Pour Jobeet, les tests se trouvent dans le sous-répertoire `test/functional/frontend/` car chaque application a son propre sous-répertoire. Ce répertoire contient déjà deux fichiers: `categoryActionsTest.php` et `jobActionsTest.php` car toutes les tâches qui génèrent un module, créent automatiquement un fichier de test fonctionnel de base :

```
// test/functional/frontend/categoryActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new sfTestFunctional(new sfBrowser());

$browser->
  get('/category/index')->

  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'index')->
  end()->

  with('response')->begin()->
    isStatusCode(200)->
    checkElement('body', '!/This is a temporary page/')->
  end()
;
```

A première vue, le script ci-dessus peut vous sembler un peu étrange. C'est parce que les méthodes de `sfBrowser` et de `sfTestFunctional` implémente une fluent interface en retournant toujours `$this`. Cela vous permet d'enchaîner les appels de méthode pour une meilleure lisibilité. L'extrait ci-dessus est équivalent à :

```
// test/functional/frontend/categoryActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');
```

```
$browser = new sfTestFunctional(new sfBrowser());

$browser->get('/category/index');
$browser->with('request')->begin();
$browser->isParameter('module', 'category');
$browser->isParameter('action', 'index');
$browser->end();

$browser->with('response')->begin();
$browser->isStatusCode(200);
$browser->checkElement('body', '!/This is a temporary page/');
$browser->end();
```

Les tests sont exécutés dans un bloc de contexte de testeur. Un bloc de contexte de testeur commence par `with('TESTER NAME')->begin()` et se termine avec `end()` :

```
$browser->
  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'index')->
  end()
;
```

Le code teste que le paramètre `module` de la requête est égal à `category` et `action` est égal à `index`.

Lorsque vous avez seulement besoin d'appeler une méthode de test sur un testeur, vous n'avez pas besoin de créer un bloc: `with('request')->isParameter('module', 'category')`

Le testeur de requête

Le **testeur de requête** fournit des méthodes de testeur pour introspecter et tester l'objet `sfWebRequest` :

Method	Description
`isParameter()`	Vérifie la valeur du paramètre de la requête
`isFormat()`	Vérifie le format de la requête
`isMethod()`	Vérifie la méthode
`hasCookie()`	Vérifie si la requête a un cookie avec
	le nom donné
`isCookie()`	Vérifie la valeur d'un cookie

Le testeur de réponse

Il y a aussi une classe **testeur de réponse** qui fournit des méthodes de testeur pour l'objet `sfWebResponse` :

Method	Description
`checkElement()`	Vérifie si le sélecteur CSS d'une réponse correspond à certains critères
`checkForm()`	Vérifie un objet de formulaire `sfForm`
`debug()`	Affiche le contenu de la réponse afin de faciliter le débogage
`matches()`	Teste une réponse avec une expression régulière
`isHeader()`	Vérifie la valeur de l'entête
`isStatusCode()`	Vérifie le code du statut de la réponse
`isRedirected()`	Vérifie si la réponse actuelle est une redirection
`isValid()`	Vérifie si une réponse est bien formée en XML (vous validez aussi la réponse en plus de son type de document en passant `true` comme argument)

Nous allons décrire plusieurs classes de testeurs dans les prochains jours (pour les

formulaires, l'utilisateur, le cache, ...).

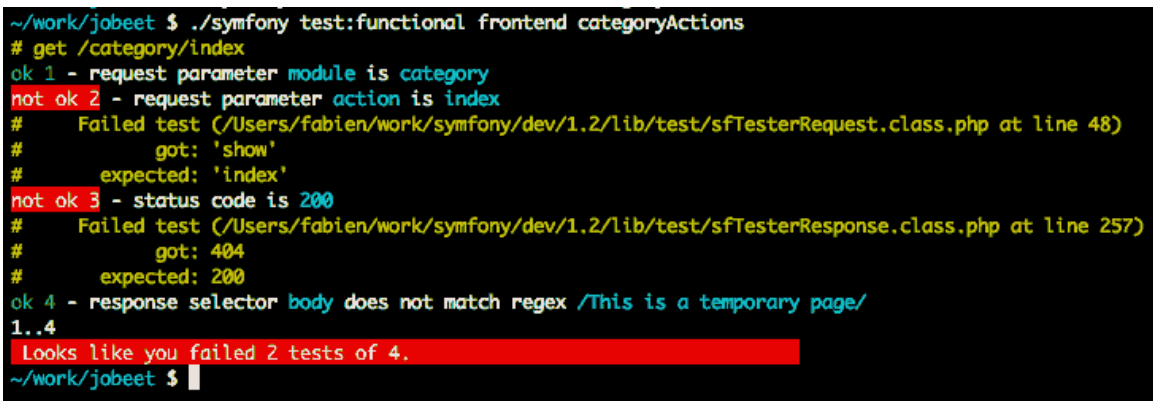
Exécution des tests fonctionnels

Comme pour les tests unitaires, le lancement des tests fonctionnels peut être fait en exécutant le fichier de test directement :

```
$ php test/functional/frontend/categoryActionsTest.php
```

Ou en utilisant la tâche `test:functional` :

```
$ php symfony test:functional frontend categoryActions
```



Données de test

Comme pour les tests unitaires Doctrine, nous avons besoin de charger des données de test, chaque fois que nous lançons un test fonctionnel. On peut réutiliser le code que nous avons écrit hier :

```
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new sfTestFunctional(new sfBrowser());
Doctrine_Core::loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

Le chargement des données dans un test fonctionnel est un peu plus facile que pour les tests unitaires car la base de données a déjà été initialisée par le script de démarrage.

Comme pour les tests unitaires, nous n'allons pas copier et coller cet extrait de code dans chaque fichier de test, mais nous allons plutôt créer notre propre classe fonctionnelle qui hérite de `sfTestFunctional` :

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function loadData()
  {
    Doctrine_Core::loadData(sfConfig::get('sf_test_dir').'/fixtures');

    return $this;
  }
}
```

Ecriture des tests fonctionnels

Ecrire des tests fonctionnels, c'est comme jouer un scénario dans un navigateur. Nous avons déjà écrit tous les scénarios que nous avons besoin de tester lors des histoires de la journée 2.

D'abord, nous allons tester la page d'accueil Jobeet en éditant le fichier de test `jobActionsTest.php`. Remplacez le code par ce qui suit :

Les emplois expirés ne sont pas affichés

```
// test/functional/frontend/jobActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();
```

```
$browser->info('1 - The homepage')->
  get('/')->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'index')->
  end()->
  with('response')->begin()->
    info('  1.1 - Expired jobs are not listed')->
    checkElement('.jobs td.position:contains("expired")', false)->
  end()
;
```

Comme avec `lime`, un message d'information peut être insérée en appelant la méthode `info()` pour rendre la sortie plus lisible. Pour vérifier l'exclusion des emplois expirés depuis la page d'accueil, nous vérifions que le sélecteur CSS `.jobs td.position:contains("expired")` ne correspond pas nulle part dans le contenu HTML de la réponse (n'oubliez pas que dans les fichiers fixtures, le seul emploi que nous avons expiré contient "expired" dans la position). Lorsque le deuxième argument de la méthode `checkElement()` est à 'true', la méthode teste l'existence de noeuds qui correspondent au sélecteur CSS.

La méthode `checkElement()` est capable d'interpréter la plupart des sélecteurs CSS3 valides.

Seulement n emplois sont affichés pour une catégorie

Ajoutez le code suivant à la fin du fichier de test :

```
// test/functional/frontend/jobActionsTest.php
$max = sfConfig::get('app_max_jobs_on_homepage');

$browser->info('1 - The homepage')->
  get('/')->
  info(sprintf('  1.2 - Only %s jobs are listed for a category', $max))->
  with('response')->
    checkElement('.category_programming tr', $max)
;
```

La méthode `checkElement()` peut également vérifier que le sélecteur CSS correspond à 'n' noeuds dans le document en passant un entier comme second argument.

Une catégorie a un lien vers la page catégorie seulement si il y a plusieurs emplois

```
// test/functional/frontend/jobActionsTest.php
$browser->info('1 - The homepage')->
  get('/')->
  info('  1.3 - A category has a link to the category page only if too many jobs')->
  with('response')->begin()->
    checkElement('.category_design .more_jobs', false)->
    checkElement('.category_programming .more_jobs')->
  end()
;
```

Dans ces tests, nous vérifions qu'il n'y a pas de lien "more jobs" pour la catégorie design (`.category_design .more_jobs` n'existe pas), et qu'il existe un lien "more jobs" pour la catégorie programmation (`.category_programming .more_jobs` existe).

Les emplois sont triés par date

```
$q = Doctrine_Query::create()
  ->select('j.*')
  ->from('JobeetJob j')
  ->leftJoin('j.JobeetCategory c')
  ->where('c.slug = ?', 'programming')
  ->andWhere('j.expires_at > ?', date('Y-m-d', time()))
  ->orderBy('j.created_at DESC');

$job = $q->fetchOne();

$browser->info('1 - The homepage')->
  get('/')->
  info('  1.4 - Jobs are sorted by date')->
```

```
  with('response')->begin()->
    checkElement(sprintf('.category_programming tr:first a[href*="/%d/"]', $job-
>getId()))->
  end()
;
```

Pour tester si les emplois sont effectivement triés par date, nous devons vérifier que le premier emploi figurant sur la page d'accueil est celui que nous attendons. Cela peut être fait en vérifiant que l'URL contient la clé primaire prévue. Comme la clé primaire peut changer entre les exécutions, nous avons besoin d'obtenir l'objet Doctrine du premier de la base de données.

Même si le test fonctionne tel quel, nous avons besoin de refactoriser un peu le code, car l'obtention du premier emploi de la catégorie programmation peut être réutilisé ailleurs dans nos tests. Nous ne voulons pas déplacer le code de la couche du Modèle car le code est le test spécifique. Au lieu de cela, nous allons déplacer le code, que nous avons créé plus tôt, de la classe `JobeetTestFunctional`. Cette classe se comporte comme une classe de testeur fonctionnel|Testeurs d'un domaine spécifique pour Jobeet :

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function getMostRecentProgrammingJob()
  {
    $q = Doctrine_Query::create()
      ->select('j.*')
      ->from('JobeetJob j')
      ->leftJoin('j.JobeetCategory c')
      ->where('c.slug = ?', 'programming');
    $q = Doctrine_Core::getTable('JobeetJob')->addActiveJobsQuery($q);

    return $q->fetchOne();
  }

  // ...
}
```

Vous pouvez maintenant remplacer le code de test précédent par le suivant :

```
// test/functional/frontend/jobActionsTest.php
$browser->info('1 - The homepage')->
  get('/')->
  info('  1.4 - Jobs are sorted by date')->
  with('response')->begin()->
    checkElement(sprintf('.category_programming tr:first a[href*="/%d/"]',
      $browser->getMostRecentProgrammingJob()->getId()))->
  end()
;
```

Chaque emploi sur la page d'accueil est cliquable

```
$job = $browser->getMostRecentProgrammingJob();

$browser->info('2 - The job page')->
  get('/')->

  info('  2.1 - Each job on the homepage is clickable and give detailed
information')->
  click('Web Developer', array(), array('position' => 1))->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'show')->
    isParameter('company_slug', $job->getCompanySlug())->
    isParameter('location_slug', $job->getLocationSlug())->
    isParameter('position_slug', $job->getPositionSlug())->
    isParameter('id', $job->getId())->
  end()
;
```

Pour tester le lien d'un emploi sur la page d'accueil, nous simulons un clic sur le text "Web Developer". Comme il y en a plusieurs sur la page, nous avons explicitement demandé au navigateur de cliquer sur le premier (`array('position' => 1)`).

Chaque paramètre de la requête est ensuite testée pour s'assurer que le routage a fait son travail correctement.

Apprendre par l'exemple

Dans cette section, nous avons fourni tout le code nécessaire pour tester les pages emploi et catégorie. Lisez le code avec soin car vous apprendrez quelques nouveaux trucs :

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function loadData()
  {
    Doctrine_Core::loadData(sfConfig::get('sf_test_dir').'/fixtures');

    return $this;
  }

  public function getMostRecentProgrammingJob()
  {
    $q = Doctrine_Query::create()
      ->select('j.*')
      ->from('JobeetJob j')
      ->leftJoin('j.JobeetCategory c')
      ->where('c.slug = ?', 'programming');
    $q = Doctrine_Core::getTable('JobeetJob')->addActiveJobsQuery($q);

    return $q->fetchOne();
  }

  public function getExpiredJob()
  {
    $q = Doctrine_Query::create()
      ->from('JobeetJob j')
      ->where('j.expires_at < ?', date('Y-m-d', time()));

    return $q->fetchOne();
  }
}

// test/functional/frontend/jobActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->info('1 - The homepage')->
  get('/')->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'index')->
  end()->
  with('response')->begin()->
    info('  1.1 - Expired jobs are not listed')->
    checkElement('.jobs td.position:contains("expired")', false)->
  end()
;

$max = sfConfig::get('app_max_jobs_on_homepage');

$browser->info('1 - The homepage')->
  info(sprintf('  1.2 - Only %s jobs are listed for a category', $max))->
  with('response')->
    checkElement('.category_programming tr', $max)
;

$browser->info('1 - The homepage')->
  get('/')->
  info('  1.3 - A category has a link to the category page only if too many jobs')->
  with('response')->begin()->
    checkElement('.category_design .more_jobs', false)->
    checkElement('.category_programming .more_jobs')->
```

```
  end()
;

$browser->info('1 - The homepage')->
  info('  1.4 - Jobs are sorted by date')->
  with('response')->begin()->
    checkElement(sprintf('.category_programming tr:first a[href*="/%d/"]', $browser-
>getMostRecentProgrammingJob()->getId()))->
  end()
;

$job = $browser->getMostRecentProgrammingJob();

$browser->info('2 - The job page')->
  get('/')->

  info('  2.1 - Each job on the homepage is clickable and give detailed
information')->
  click('Web Developer', array(), array('position' => 1))->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'show')->
    isParameter('company_slug', $job->getCompanySlug())->
    isParameter('location_slug', $job->getLocationSlug())->
    isParameter('position_slug', $job->getPositionSlug())->
    isParameter('id', $job->getId())->
  end()->

  info('  2.2 - A non-existent job forwards the user to a 404')->
  get('/job/foo-inc/milano-italy/0/painter')->
  with('response')->isStatusCode(404)->

  info('  2.3 - An expired job page forwards the user to a 404')->
  get(sprintf('/job/sensio-labs/paris-france/%d/web-developer', $browser-
>getExpiredJob()->getId()))->
  with('response')->isStatusCode(404)
;

// test/functional/frontend/categoryActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->info('1 - The category page')->
  info('  1.1 - Categories on homepage are clickable')->
  get('/')->
  click('Programming')->
  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'show')->
    isParameter('slug', 'programming')->
  end()->

  info(sprintf('  1.2 - Categories with more than %s jobs also have a "more" link',
sfConfig::get('app_max_jobs_on_homepage')))->
  get('/')->
  click('27')->
  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'show')->
    isParameter('slug', 'programming')->
  end()->

  info(sprintf('  1.3 - Only %s jobs are listed',
sfConfig::get('app_max_jobs_on_category')))->
  with('response')->checkElement('.jobs tr',
sfConfig::get('app_max_jobs_on_category'))->

  info('  1.4 - The job listed is paginated')->
  with('response')->begin()->
    checkElement('.pagination_desc', '/32 jobs/')->
```

```
    checkElement('.pagination_desc', '#page 1/2#')->
  end()->

  click('2')->
  with('request')->begin()->
    isParameter('page', 2)->
  end()->
  with('response')->checkElement('.pagination_desc', '#page 2/2#')
;
```

Débogage des tests fonctionnels

Parfois, un test fonctionnel échoue. Comme symfony simule un navigateur, sans aucune interface graphique, cela peut être difficile pour diagnostiquer le problème. Heureusement, symfony prévoit la méthode ~debug|Debogage~() pour afficher l'entête de la réponse et le contenu :

```
$browser->with('response')->debug();
```

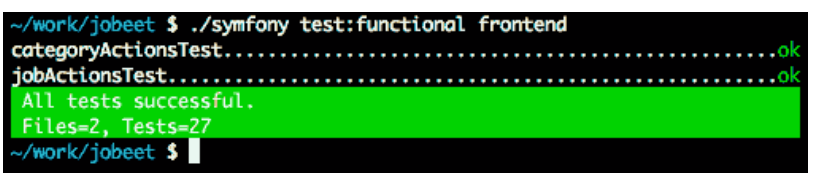
La méthode debug() peut être inséré n'importe où dans un bloc de testeur de response et mettra fin à l'exécution de script.

Validation des tests fonctionnels

La tâche `test:functional` peut aussi être utilisée pour lancer tous les tests fonctionnels d'une application :

```
$ php symfony test:functional frontend
```

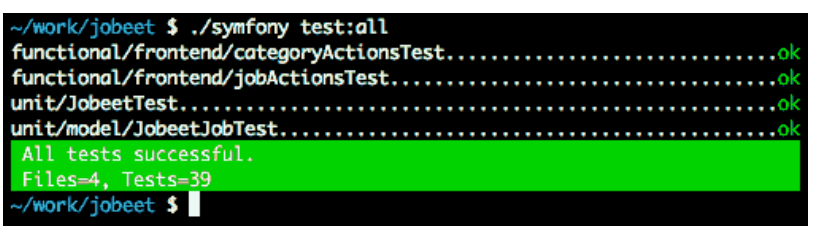
La tâche renvoie une seule ligne pour chaque fichier de test :



Validation des tests

Comme vous pouvez vous y attendre, il existe aussi une tâche pour lancer tous les tests pour un projet (unitaires et fonctionnels) :

```
$ php symfony test:all
```



Quand vous avez une grande série de test, cela peut prendre beaucoup de temps pour lancer tous les tests à chaque fois que vous effectuez un changement, surtout si certains tests échouent. Car chaque fois que vous fixez un test, vous devez exécuter la suite des tests à nouveau pour vous assurer que vous n'avez pas cassé autre chose. Mais tant que ces tests ne sont pas fixés, il est inutile de re-exécuter toutes les autres tests. La tâche `test:all` a une option `--only-failed` qui force la tâche à ré-exécuter seulement les tests qui ont échoué au cours de l'exécution précédente :

```
$ php symfony test:all --only-failed
```

La première fois que vous exécutez la tâche, tous les tests sont exécutés comme d'habitude. Mais pour les séries de tests suivants, seuls les tests qui ont échoués la dernière fois sont exécutés. Comme vous corriger votre code, certains tests vont passer, et seront supprimés lors des prochains passages. Lorsque tous les tests passent, la suite complète des tests est exécutée ... Vous pouvez ensuite nettoyer et recommencer.

Si vous souhaitez intégrer votre suite de tests dans un processus d'intégration continue, utilisez l'option `--xml` pour forcer la tâche `test:all` à générer une sortie XML compatible

JUnit.

```
$ php symfony test:all --xml=log.xml
```

À demain

Cela termine notre tour des outils de test de symfony. Vous n'avez aucune excuse pour ne plus tester vos applications ! Avec le framework lime et le framework de test fonctionnel, symfony fournit des outils puissants pour vous aider à écrire des tests avec peu d'effort.

Nous n'avons fait que gratter la surface des tests fonctionnels. A partir de maintenant, chaque fois que nous mettrons en place une fonctionnalité, nous allons aussi écrire les tests pour apprendre plus de fonctionnalités du framework de test.

Demain, nous parlerons encore d'une autre grande caractéristique de symfony: le framework de formulaire.

« Jour 8 : Les tests unitaires

Jour 10 : Les formulaires »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 10 : Les formulaires

Le chapitre précédent du tutoriel de Jobeet a pris un bon départ avec l'introduction du framework de test de symfony. Nous allons continuer, dans ce nouveau chapitre, avec le framework de formulaire

Le framework de formulaire

Tout site Web a des formulaires : du simple formulaire de contact à celui avec beaucoup plus de champs. L'écriture de formulaire est aussi une des tâches les plus complexes et l'une des plus fastidieuses pour un développeur web : vous devez écrire le formulaire HTML, implémenter des règles de validation pour chaque champ, traiter les valeurs pour les stocker dans une base de données, afficher les messages d'erreur, repeupler les champs dans le cas d'erreurs, et bien plus encore ...

Bien sûr, au lieu de réinventer la roue, encore et encore, symfony fournit un framework pour faciliter la gestion de formulaire. Le framework de formulaire est composé de trois parties :

- **validation**: Le sous-framework de validation fournit les classes pour valider les saisies (entier, chaine, adresse email, ...)
- **widgets**: Le sous-framework de widget fournit les classes pour la production des champs HTML (input, textarea, select, ...)
- **forms**: Les classes formulaire représentent des formulaires faits de widgets et de validateurs et fournit des méthodes pour aider la gestion du formulaire. Chaque champs du formulaire a son propre validateur et son propre widget.

Les formulaires

Un formulaire de symfony est une classe composé de champs. Chaque champ a un nom, un validateur et un widget. Un simple `ContactForm` peut être défini avec la classe suivante :

```
class ContactForm extends sfForm
{
  public function configure()
  {
    $this->setWidgets(array(
      'email'   => new sfWidgetFormInputText(),
      'message' => new sfWidgetFormTextarea(),
    ));

    $this->setValidators(array(
      'email'   => new sfValidatorEmail(),
      'message' => new sfValidatorString(array('max_length' => 255)),
    ));
  }
}
```

Les champs du formulaire sont configurés dans la méthode `configure()`, en utilisant les méthodes `setValidators()` et `setWidgets()`.

Le frameeework de formulaire est livré avec un lot de widgets et validators. L'API les décrit assez largement avec toutes les options, les erreurs et les messages d'erreur par défaut.

Les noms des classes des widgets et des validateurs sont assez explicites : le champ `email` sera rendu par une balise HTML `<input>` (`sfWidgetFormInputText`) et sera validé par une adresse email (`sfValidatorEmail`). Le champ `message` sera rendu par une balise `<textarea>` (`sfWidgetFormTextarea`), et doit être une chaine d'au plus 255 caractères (`sfValidatorString`).

Par défaut, tous les champs sont obligatoires, car la valeur par défaut pour l'option `required` est `true`. Ainsi, la définition de la validation pour `email` est équivalente à `new sfValidatorEmail(array('required' => true))`.

Vous pouvez fusionner un formulaire dans un autre en utilisant la méthode `mergeForm()`, ou incorporer un formulaire à l'aide de la méthode `embedForm()` :

```
$this->mergeForm(new AnotherForm());
$this->embedForm('name', new AnotherForm());
```

Les formulaires de Doctrine

La plupart du temps, un formulaire doit être sérialisé à la base de données. Comme symfony sait déjà tout sur le modèle de votre base de données, il peut générer automatiquement des formulaires basés sur ces informations. En fait, lorsque vous avez lancé la tâche `doctrine:build --all` au cours du chapitre 3, symfony a automatiquement appelé la tâche `doctrine:build --forms` :

```
$ php symfony doctrine:build --forms
```

La tâche `doctrine:build --forms` génère des classes de formulaires dans le répertoire `lib/form/`. L'organisation de ces fichiers générés est semblable à celle de `lib/model/`. Chaque classe du modèle a une classe de formulaire correspondante (par exemple `JobeetJob` a `JobeetJobForm`), qui est vide par défaut car il hérite d'une classe de base :

```
// lib/form/doctrine/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
  }
}
```

En parcourant les fichiers générés dans le sous-répertoire `lib/form/doctrine/base/`, vous pourrez voir de nombreux exemples d'usage de symfony intégré dans les widgets et les validateurs.

Vous pouvez désactiver la génération des formulaires sur certains modèles en passant des paramètres au behavior Doctrine de symfony :

```
SomeModel:
  options:
    symfony:
      form: false
      filter: false
```

Personnalisation du formulaire d'emploi

Le formulaire d'emploi est un parfait exemple pour apprendre la personnalisation du formulaires. Voyons comment le personnaliser, étape par étape.

D'abord, changez le lien "Post a Job" dans la mise en page pour être en mesure de vérifier les modifications directement dans votre navigateur :

```
<!-- apps/frontend/templates/layout.php -->
<a href="<?php echo url_for('@job_new') ?>">Post a Job</a>
```

Par défaut, un formulaire Doctrine affiche des champs pour toutes les colonnes de la table. Mais pour le formulaire d'emploi, certains d'entre eux ne doit pas être modifiable par l'utilisateur final. La suppression des champs à partir d'un formulaire est aussi simple que d'en désactiver :

```
// lib/form/doctrine/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    unset(
      $this['created_at'], $this['updated_at'],
```

```
      $this['expires_at'], $this['is_activated']
    );
  }
}
```

La désactivation d'un champ signifie que le widget et le validateur du champs sont supprimés.

Au lieu de désactiver les champs que vous ne souhaitez pas afficher, vous pouvez également explicitement lister les champs que vous souhaitez avec la méthode `useFields()` :

```
// lib/form/doctrine/JobeetJobForm.class.php
        class JobeetJobForm extends BaseJobeetJobForm
    {
      public function configure()
      {
        $this->useFields(array('category_id', 'type', 'company', 'logo', 'url',
'position', 'location', 'description', 'how_to_apply', 'token', 'is_public',
'email'));
      }
    }
```

La méthode `useFields()` fait deux choses automatiquement pour vous : elle ajoute les champs masqués et le tableau des champs est utilisée pour changer l'ordre des champs.

Répertorier de manière explicite les champs du formulaire que vous souhaitez afficher signifie que lors de l'ajout de nouveaux champs à un formulaire de base, ils n'apparaîtront pas automatiquement dans votre formulaire (pensez à un modèle de formulaire où vous ajoutez une nouvelle colonne à la table liée).

La configuration du formulaire doit parfois être plus précis que l'introspection à partir du schéma de la base de données. Par exemple, la colonne `email` est un `varchar` dans le schéma, mais nous avons besoin de cette colonne pour être validé comme un email. Changeons la valeur par défaut `sfValidatorString` par `sfValidatorEmail` :

```
// lib/form/doctrine/JobeetJobForm.class.php
public function configure()
{
  // ...

  $this->validatorSchema['email'] = new sfValidatorEmail();
}
```

Le remplacement de la validation par défaut n'est pas toujours la meilleure solution, comme les règles de validation par défaut introspectés depuis le schéma de la base de données sont perdues (`new sfValidatorString(array('max_length' => 255))`). Il est presque toujours préférable d'ajouter le nouveau validateur à celui déjà existant à l'aide du validateur spécial `sfValidatorAnd` :

```
// lib/form/doctrine/JobeetJobForm.class.php
public function configure()
{
  // ...

  $this->validatorSchema['email'] = new sfValidatorAnd(array(
    $this->validatorSchema['email'],
    new sfValidatorEmail(),
  ));
}
```

Le validateur `sfValidatorAnd` prend un tableau de validateurs qui doivent tous passer pour que cette valeur soit valide. L'astuce ici est de faire référence au validateur actuel (`$this->validatorSchema['email']`), et d'ajouter le nouveau.

Vous pouvez également utiliser le validateur `sfValidatorOr` pour forcer une valeur à passer pour au moins un validateur. Et bien sûr, vous pouvez mélanger et fusionner les validateurs `sfValidatorAnd` et `sfValidatorOr` pour créer des validateurs complexes sur base de booléen.

Même si la colonne `type` est également un `varchar` dans le schéma, nous voulons sa valeur

pour se limiter à une liste de choix : temps plein, temps partiel, ou freelance.

D'abord, nous allons définir les valeurs possibles dans `JobeetJobTable` :

```
// lib/model/doctrine/JobeetJobTable.class.php
class JobeetJobTable extends Doctrine_Table
{
  static public $types = array(
    'full-time' => 'Full time',
    'part-time' => 'Part time',
    'freelance' => 'Freelance',
  );

  public function getTypes()
  {
    return self::$types;
  }

  // ...
}
```

Puis, utilisez `sfWidgetFormChoice` pour le widget `type` :

```
$this->widgetSchema['type'] = new sfWidgetFormChoice(array(
  'choices'  => Doctrine_Core::getTable('JobeetJob')->getTypes(),
  'expanded' => true,
));
```

`sfWidgetFormChoice` représente un widget de choix qui peut être affiché par un widget différent selon certaines options de configuration (`expanded` et `multiple`) :

- Liste déroulante (<select>) : `array('multiple' => false, 'expanded' => false)`
- Boîte déroulante (<select multiple="multiple">) : `array('multiple' => true, 'expanded' => false)`
- Liste de radiobuttons : `array('multiple' => false, 'expanded' => true)`
- List de checkboxes : `array('multiple' => true, 'expanded' => true)`

Si vous voulez que l'un des radiobutton soit sélectionné par défaut (`full-time` par exemple), vous pouvez changer la valeur par défaut dans le schéma de base de données.

Même si vous pensez que personne peut soumettre un valeur non valide, un pirate peut facilement contourner les choix du widget en utilisant des outils comme curl ou la barre d'outils de développeur web de Firefox. Changeons le validateur pour restreindre les choix possibles :

```
$this->validatorSchema['type'] = new sfValidatorChoice(array(
  'choices' => array_keys(Doctrine_Core::getTable('JobeetJob')->getTypes()),
));
```

Comme la colonne `logo` va stocker le nom du fichier du logo associé à l'emploi, nous devons changer le widget en une balise file input :

```
$this->widgetSchema['logo'] = new sfWidgetFormInputFile(array(
  'label' => 'Company logo',
));
```

Pour chaque champ, symfony génère automatiquement un label (qui sera utilisé dans la balise <label> rendue). Ceci peut être modifié avec l'option `label`.

Vous pouvez également modifier les labels d'un batch avec la méthode `setLabels()` du tableau de widget :

```
$this->widgetSchema->setLabels(array(
  'category_id'    => 'Category',
  'is_public'      => 'Public?',
  'how_to_apply'   => 'How to apply?',
));
```

Nous devons également changer le validateur par défaut :

```
$this->validatorSchema['logo'] = new sfValidatorFile(array(
  'required'   => false,
  'path'       => sfConfig::get('sf_upload_dir').'/jobs',
```

```
    'mime_types' => 'web_images',
  ));
```

`sfValidatorFile` est assez intéressant car il fait un certain nombre de choses :

- Valider que le fichier téléchargé est une image dans un format web (`mime_types`)
- Renomme le fichier en quelque chose d'unique
- Stocke le fichier dans le `path` donné
- Met à jour la colonne `logo` avec le nom généré

Vous devez créer le répertoire logo (`web/uploads/jobs/`) et vérifier qu'il est accessible en écriture par le serveur web.

Comme le validateur enregistrera le chemin relatif dans la base de données, changer le chemin utilisé dans le template `showSuccess` :

```
// apps/frontend/modules/job/templates/showSuccess.php
<img src="/uploads/jobs/<?php echo $job->getLogo() ?>" alt="<?php echo $job-
>getCompany() ?> logo" />
```

Si une méthode `generateLogoFilename()` existe dans le modèle, il sera appelé par le validateur et le résultat remplacera le nom de fichier `logo` généré par défaut. La méthode prend l'objet `sfValidatedFile` comme argument.

Tout comme vous pouvez remplacer le label généré de n'importe quel champ, vous pouvez aussi définir un message d'aide. Ajoutons en un pour la colonne `is_public` pour mieux expliquer sa signification :

```
$this->widgetSchema->setHelp('is_public', 'Whether the job can also be published on
affiliate websites or not.');
```

La classe `JobeetJobForm` finale se lit comme suit :

```
// lib/form/doctrine/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['expires_at'], $this['is_activated']
    );

    $this->validatorSchema['email'] = new sfValidatorAnd(array(
      $this->validatorSchema['email'],
      new sfValidatorEmail(),
    ));

    $this->widgetSchema['type'] = new sfWidgetFormChoice(array(
      'choices'  => Doctrine_Core::getTable('JobeetJob')->getTypes(),
      'expanded' => true,
    ));
    $this->validatorSchema['type'] = new sfValidatorChoice(array(
      'choices' => array_keys(Doctrine_Core::getTable('JobeetJob')->getTypes()),
    ));

    $this->widgetSchema['logo'] = new sfWidgetFormInputFile(array(
      'label' => 'Company logo',
    ));

    $this->widgetSchema->setLabels(array(
      'category_id'    => 'Category',
      'is_public'      => 'Public?',
      'how_to_apply'   => 'How to apply?',
    ));

    $this->validatorSchema['logo'] = new sfValidatorFile(array(
      'required'   => false,
```

```
      'path'       => sfConfig::get('sf_upload_dir').'/jobs',
      'mime_types' => 'web_images',
    ));

    $this->widgetSchema->setHelp('is_public', 'Whether the job can also be published
on affiliate websites or not.');
  }
}
```

Le template du formulaire

Maintenant que la classe de formulaire a été personnalisé, nous avons besoin de l'afficher. Le Template du formulaire est le même, que vous vouliez créer un nouvel emploi ou modifier un existant. En fait, les deux Templates newSuccess.php et editSuccess.phpsont assez similaires:

```
<!-- apps/frontend/modules/job/templates/newSuccess.php -->
<?php use_stylesheet('job.css') ?>

<h1>Post a Job</h1>

<?php include_partial('form', array('form' => $form)) ?>
```

Si vous n'avez pas ajouté de la feuille de style job pour le moment, il est temps de le faire dans les deux Templates (`<?php use_stylesheet('job.css') ?>`).

Le formulaire est lui-même rendu dans le partial `_form`. Remplacez le contenu du partial généré `_form` par le code suivant :

```
<!-- apps/frontend/modules/job/templates/_form.php -->
<?php use_stylesheets_for_form($form) ?>
<?php use_javascripts_for_form($form) ?>

<?php echo form_tag_for($form, '@job') ?>
  <table id="job_form">
    <tfoot>
      <tr>
        <td colspan="2">
          <input type="submit" value="Preview your job" />
        </td>
      </tr>
    </tfoot>
    <tbody>
      <?php echo $form ?>
    </tbody>
  </table>
</form>
```

Les helpers `use_javascripts_for_form()` et `use_stylesheets_for_form()` incluent des dépendances de JavaScript et de feuille de style nécessaires pour les widgets du formulaire.

Même si le formulaire emploi n'a pas besoin de JavaScript ou d'un fichier de style, c'est une bonne habitude de garder l'appels de ces helpers "juste au cas où". Il peut sauver votre journée plus tard, si vous décidez de changer un widget qui a besoin de quelques JavaScript ou d'une feuille de style spécifique.

Le helper `form_tag_for()` génère une balise `<form>` pour le formulaire et la route donnés et modifie les méthodes HTTP en POST ou en PUT PUT selon si l'objet est nouveau ou non. Il s'occupe aussi de l'attribut `multipart` si le formulaire a aucune balise input file.

Finalement, le `<?php echo $form ?>` rend les widgets du formulaire.

Personnalisation de l'aspect d'un formulaire

Par défaut, le `<?php echo $form ?>` rend les widgets du formulaire en tant que lignes de tableau.

La plupart du temps, vous aurez besoin de personnaliser la présentation de vos formulaires. L'objet du formulaire fournit de nombreuses méthodes utiles pour cette personnalisation :

Méthode	Description
`render()`	Rend le formulaire (equivalent à la sortie de `echo $form`)
`renderHiddenFields()`	Rend les champs masqués
`hasErrors()`	Retourne `true` si le formulaire a quelques erreurs
`hasGlobalErrors()`	Retourne `true` si le formulaire a des erreurs globales
`getGlobalErrors()`	Retourne un tableau des erreurs globales
`renderGlobalErrors()`	Rend les erreurs globales

Le formulaire se comporte aussi comme un tableau de champs. Vous pouvez accéder au champ company avec `$form['company']`. L'objet retourné fournit des méthodes pour rendre chaque élément du champs :

Méthode	Description
`renderRow()`	Rend la ligne du champ
`render()`	Rend le widget du champ
`renderLabel()`	Rend le label du champ
`renderError()`	Rend les messages d'érreur du champ s'il y'en a
`renderHelp()`	Rend le message d'aide du champ

L'instruction `echo $form` est équivalent à :

```
<?php foreach ($form as $widget): ?>
  <?php echo $widget->renderRow() ?>
<?php endforeach ?>
```

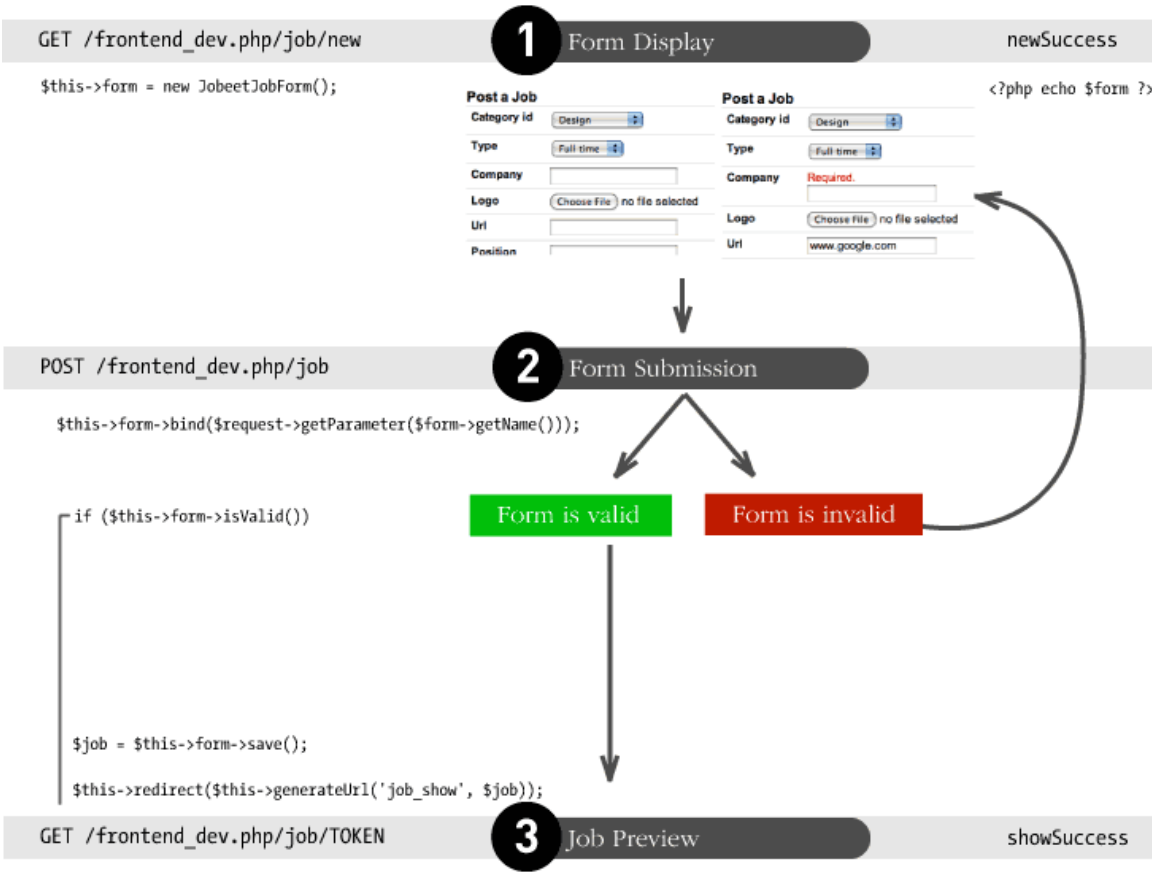
L'action du formulaire

Nous avons maintenant une classe de formulaire et un template qui fait le rendu. Maintenant, il est temps de le faire réellement fonctionner avec certaines actions.

Le formulaire emploi est géré par cinq méthodes dans le module `job` :

- **new**: Affiche un formulaire non renseigné pour créer un nouvel emploi
- **edit**: Affiche un formulaire pour modifier un emploi existant
- **create**: Crée un nouvel emploi avec les valeurs soumises par l'utilisateur
- **update**: Met à jour un emploi avec les valeurs soumises par l'utilisateur
- **processForm**: Appelé par `create` et `update`, il traite le formulaire (validation, repopulation du formulaire et sérialisation vers la base de données)

Tous les formulaires ont le cycle de vie suivant :



Comme nous avons créé une collection de route Doctrine il y a 5 jours pour le module `job`, nous pouvons simplifier le code pour les méthodes de gestion de formulaire :

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeNew(sfWebRequest $request)
{
  $this->form = new JobeetJobForm();
}

public function executeCreate(sfWebRequest $request)
{
  $this->form = new JobeetJobForm();
  $this->processForm($request, $this->form);
  $this->setTemplate('new');
}

public function executeEdit(sfWebRequest $request)
{
  $this->form = new JobeetJobForm($this->getRoute()->getObject());
}

public function executeUpdate(sfWebRequest $request)
{
  $this->form = new JobeetJobForm($this->getRoute()->getObject());
  $this->processForm($request, $this->form);
  $this->setTemplate('edit');
}

public function executeDelete(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $job->delete();

  $this->redirect('job/index');
```

```
}

protected function processForm(sfWebRequest $request, sfForm $form)
{
  $form->bind(
    $request->getParameter($form->getName()),
    $request->getFiles($form->getName())
  );

  if ($form->isValid())
  {
    $job = $form->save();

    $this->redirect('job_show', $job);
  }
}
```

Lorsque vous naviguez vers la page `/job/new`, une nouvelle instance du formulaire est créé et transmise au Template (action new).

Lorsque l'utilisateur soumet le formulaire (action `create`), le formulaire est lié (méthode `bind()`) avec les valeurs soumises par l'utilisateur et la validation est déclenchée.

Une fois que le formulaire est lié, il est possible de vérifier sa validité en utilisant la méthode `isValid()` : si le formulaire est valide (elle retourne `true`), l'emploi est sauvé dans la base de données (`$form->save()`), et l'utilisateur est redirigé vers la page de prévisualisation d'emploi, sinon, le template `newSuccess.php` s'affiche à nouveau avec les valeurs soumises par l'utilisateur et les messages d'erreur associés.

La méthode `setTemplate()` change le Template utilisé pour une action donnée. Si le formulaire soumis n'est pas valide, les méthodes `create` et `update` utilisent le même template respectif que l'action new et `edit` pour ré-afficher le formulaire avec les messages d'erreur.

La modification d'un emploi existant est assez similaire. La seule différence entre l'action new et `edit` est que l'objet job à modifier, est passé comme premier argument au constructeur du formulaire. Cet objet sera utilisé pour les valeurs par défaut de widget dans le Template (les valeurs par défaut sont un objet pour le formulaire Doctrine, mais un simple tableau pour des formulaires simples).

Vous pouvez également définir des valeurs par défaut pour le formulaire de création. Une solution est de déclarer les valeurs dans le schéma de base de données. Une autre est de passer un objet Job pré-modifié au constructeur du formulaire.

Changer la méthode `executeNew()` pour définir `full-time` comme étant la valeur par défaut pour la colonne `type` :

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeNew(sfWebRequest $request)
{
  $job = new JobeetJob();
  $job->setType('full-time');

  $this->form = new JobeetJobForm($job);
}
```

Lorsque le formulaire est lié, les valeurs par défaut sont remplacées par celles soumises par l'utilisateur. Les valeurs soumises par l'utilisateur seront utilisées pour le repeuplement du formulaire lorsque ce dernier est à nouveau affichée en cas d'erreurs de validation.

Protection du formulaire d'emploi avec un jeton (token)

Tout doit fonctionner correctement maintenant. Pour l'instant, l'utilisateur doit saisir le jeton pour l'emploi. Mais le jeton de l'emploi doit être généré automatiquement quand un nouvel emploi est créé, car nous ne pouvons pas se fier à l'utilisateur pour fournir un jeton unique.

Mettez à jour la méthode `save()` de JobeetJob pour ajouter la logique qui génère le jeton avant qu'un nouvel emploi soit enregistré :

```
// lib/model/doctrine/JobeetJob.class.php
```

```
public function save(Doctrine_Connection $conn = null)
{
  // ...

  if (!$this->getToken())
  {
    $this->setToken(sha1($this->getEmail().rand(11111, 99999)));
  }

  return parent::save($conn);
}
```

Vous pouvez maintenant supprimer le champ `token` du formulaire :

```
// lib/form/doctrine/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['expires_at'], $this['is_activated'],
      $this['token']
    );

    // ...
  }

  // ...
}
```

Si vous vous souvenez des histoires d'utilisateurs du jour 2, un emploi peut être modifié que si l'utilisateur connaît le jeton associé. À l'heure actuelle, il est assez facile de modifier ou de supprimer n'importe quel emploi, en essayant juste de deviner l'URL. Car l'URL de edit est `/job/ID/edit`, où `ID` est la clé primaire de l'emploi.

Par défaut, une route `sfDoctrineRouteCollection` génère des URL avec la clé primaire, mais elle peut être changée par n'importe quelle colonne unique en faisant passer l'option `column` :

```
# apps/frontend/config/~routing|Routing~.yml
job:
  class:        sfDoctrineRouteCollection
  options:      { model: JobeetJob, column: token }
  requirements: { token: \w+ }
```

Notez que nous avons également modifié le paramètre requirements `token` pour faire correspondre n'importe quelle chaîne car le requirements de symfony par défaut est `\d+` pour la clé unique.

Maintenant, toutes les routes liées aux emplois, à l'exception des `job_show_user`, incorporent le jeton. Par exemple, la route pour modifier un emploi est maintenant sur le modèle suivant :

```
http://www.jobeet.com.localhost/job/TOKEN/edit
```

Vous aurez aussi besoin de changer le lien "Edit" dans le Template `showSuccess` :

```
<!-- apps/frontend/modules/job/templates/showSuccess.php -->
<a href="<?php echo url_for('job_edit', $job) ?>">Edit</a>
```

La page de prévisualisation

La page de prévisualisation est la même que l'affichage de la page emploi. Grâce au routage, si l'utilisateur arrive avec le bon jeton, il sera accessible dans le paramètre de la requête `token`.

Si l'utilisateur entre dans l'URL sous forme de jeton, nous allons ajouter une barre d'admininistrateur en haut. Au début du Template `showSuccess`, ajoutez un partial pour mettre la barre d'administrateur et supprimer le lien `edit` en bas :

```
<!-- apps/frontend/modules/job/templates/showSuccess.php -->
<?php if ($sf_request->getParameter('token') == $job->getToken()): ?>
  <?php include_partial('job/admin', array('job' => $job)) ?>
<?php endif ?>
```

Ensuite, créez le partial `_admin` :

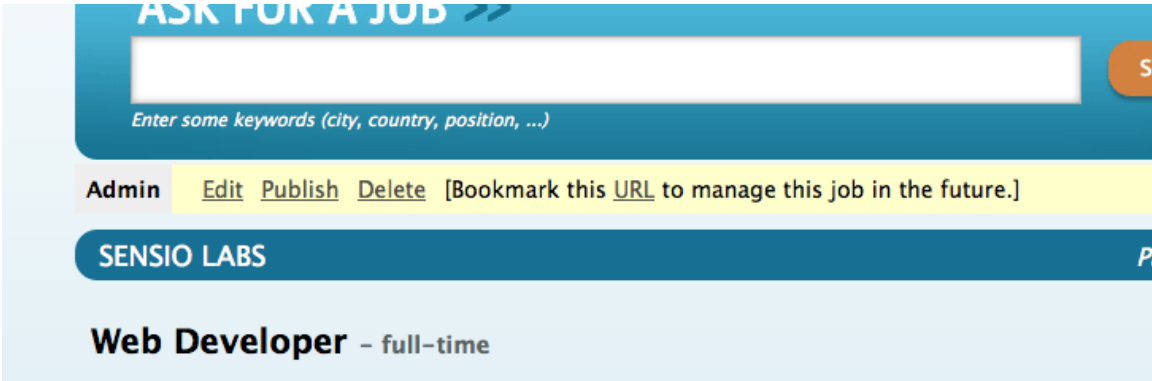
```
<!-- apps/frontend/modules/job/templates/_admin.php -->
<div id="job_actions">
  <h3>Admin</h3>
  <ul>
    <?php if (!$job->getIsActivated()): ?>
      <li><?php echo link_to('Edit', 'job_edit', $job) ?></li>
      <li><?php echo link_to('Publish', 'job_edit', $job) ?></li>
    <?php endif ?>
    <li><?php echo link_to('Delete', 'job_delete', $job, array('method' => 'delete',
'confirm' => 'Are you sure?')) ?></li>
    <?php if ($job->getIsActivated()): ?>
      <li<?php $job->expiresSoon() and print ' class="expires_soon"' ?>>
        <?php if ($job->isExpired()): ?>
          Expired
        <?php else: ?>
          Expires in <strong><?php echo $job->getDaysBeforeExpires() ?></strong> days
        <?php endif ?>

        <?php if ($job->expiresSoon()): ?>
         - <a href="">Extend</a> for another <?php echo
sfConfig::get('app_active_days') ?> days
        <?php endif ?>
      </li>
    <?php else: ?>
      <li>
        [Bookmark this <?php echo link_to('URL', 'job_show', $job, true) ?> to manage
this job in the future.]
      </li>
    <?php endif ?>
  </ul>
</div>
```

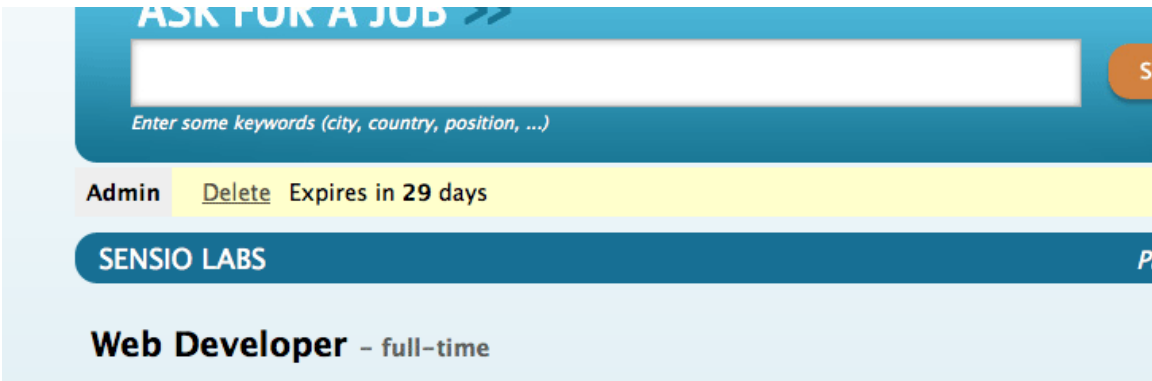
Il y a beaucoup de code, mais la plupart du code est simple à comprendre.

Pour rendre le Template plus lisible, nous avons ajouté un ensemble de raccourci de méthodes dans la classe JobeetJob :

```
// lib/model/doctrine/JobeetJob.class.php
public function getTypeName()
{
  $types = Doctrine_Core::getTable('JobeetJob')->getTypes();
  return $this->getType() ? $types[$this->getType()] : '';
}

public function isExpired()
{
  return $this->getDaysBeforeExpires() < 0;
}

public function expiresSoon()
{
  return $this->getDaysBeforeExpires() < 5;
}

public function getDaysBeforeExpires()
{
  return ceil(($this->getDateTimeObject('expires_at')->format('U') - time()) / 86400);
}
```

a barre d'administrateur affiche les différentes actions en fonction de l'état de l'emploi :





Vous serez capable de voir la barre "activée" après la section suivante.

Activation et Publication d'un emploi

Dans la section précédente, il y a un lien pour publier l'emploi. Le lien doit être changé pour pointer vers une nouvelle action `publish`. Au lieu de créer une nouvelle route, on peut juste configurer la route `job` existante :

```
# apps/frontend/config/routing.yml
job:
  class:   sfDoctrineRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: put }
  requirements:
    token: \w+
```

Le `object_actions` prend un tableau des actions supplélentaires pour un objet donné. Nous pouvons maintenant modifier le lien du lien "Publish" :

```
<!-- apps/frontend/modules/job/templates/_admin.php -->
<li>
  <?php echo link_to('Publish', 'job_publish', $job, array('method' => 'put')) ?>
</li>
```

La dernière étape est de créer l'action `publish` :

```
// apps/frontend/modules/job/actions/actions.class.php
public function executePublish(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $job->publish();

  $this->getUser()->setFlash('notice', sprintf('Your job is now online for %s days.',
sfConfig::get('app_active_days')));

  $this->redirect('job_show_user', $job);
}
```

Le lecteur attentif aura remarqué que le lien "Publish" est soumis avec la méthode HTTP put. Pour simuler la méthode put, le lien est automatiquement converti en un formulaire lorsque vous cliquez dessus.

Et comme nous avons activé la protection CSRF, le helper `link_to()` embarque un jeton CSRF dans le lien et la méthode `checkCSRFProtection()` de l'objet de la requête vérifie la validité de celui-ci lors de la soumission.

La méthode `executePublish()` utilise une nouvelle méthode `publish()` qui peut être définie comme suit :

```
// lib/model/doctrine/JobeetJob.class.php
public function publish()
{
  $this->setIsActivated(true);
  $this->save();
}
```

Vous pouvez maintenant tester la nouvelle fonctionnalité de publication dans votre navigateur.

Mais nous avons encore quelque chose à corriger. Les emplois non-activés ne doivent pas être accessibles. Ce qui signifie qu'ils ne doivent pas apparaître sur la page d'accueil Jobeet et ne doivent pas être accessibles par leur URL. Comme nous avons créé une méthode `addActiveJobsQuery()` pour limiter une `Doctrine_Query` aux emplois actifs, nous pouvons le modifier et ajouter des nouvelles exigences à la fin :

```
// lib/model/doctrine/JobeetJobTable.class.php
public function addActiveJobsQuery(Doctrine_Query $q = null)
{
  // ...

  $q->andWhere($alias . '.is_activated = ?', 1);

  return $q;
}
```

C'est tout. Vous pouvez maintenant le tester dans votre navigateur. Tous les emplois non-activés ont disparu de la page d'accueil, même si vous connaissez son URL, ils ne sont plus accessibles. Ils sont cependant accessibles si l'on connait URL du jeton de l'emploi. Dans ce cas, l'aperçu de l'emploi sera affichée avec la barre d'administration.

C'est un des grands avantages du modèle MVC et la refactorisation nous a fait faire du chemin. Un seul changement dans une seule méthode a été nécessaire pour ajouter la nouvelle exigence.

Lorsque nous avons créé la méthode `getWithJobs()`, nous avons oublié d'utiliser la méthode `addActiveJobsQuery()`. Donc, nous avons besoin de la modifier et ajouter la nouvelle exigence :

```
class JobeetCategoryTable extends Doctrine_Table
{
  public function getWithJobs()
  {
    // ...

    $q->andWhere('j.is_activated = ?', 1);

    return $q->execute();
  }
```

Conclusion

Ce chapitre a été composé avec beaucoup de nouvelles informations, mais nous espérons que vous avez maintenant une meilleure compréhension du framework de formulaire de symfony.

Nous savons que certains d'entre vous ont remarqué que nous avons oublié quelque chose aujourd'hui ... Nous n'avons pas mis en œuvre de test pour les nouvelles fonctionnalités. Parce que l'écriture de test est une partie importante du développement d'une application, c'est la première chose que nous ferons dans le prochain chapitre.

« Jour 9 : Les tests fonctionnels

Jour 11 : Testez votre formulaire »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

the SENSIOLABS network

Since 1998, Sensio Labs has been promoting the Open-Source software movement by providing quality web application development, training, consulting.
Sensio Labs also supports several large Open-Source projects.

Open-Source Products

Symfony - MVC framework
Symfony Components
Doctrine - ORM
Swift Mailer - Mailing library
Twig - Template library
Pirum - PEAR channel server

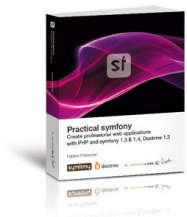
Servic

Training
Guru -
Partner
Books -
Confere
Confere

Practical symfony
Jour 11 : Testez votre formulaire

Dans le chapitre 10, nous avons créé notre premier formulaire avec symfony. Les gens sont maintenant en mesure de publier un nouvel emploi dans Jobeet mais nous avons manqué de temps pour ajouter quelques tests.

C'est ce que nous ferons dans ce chapitre. En chemin, nous en apprendrons également plus sur le framework de formulaire.

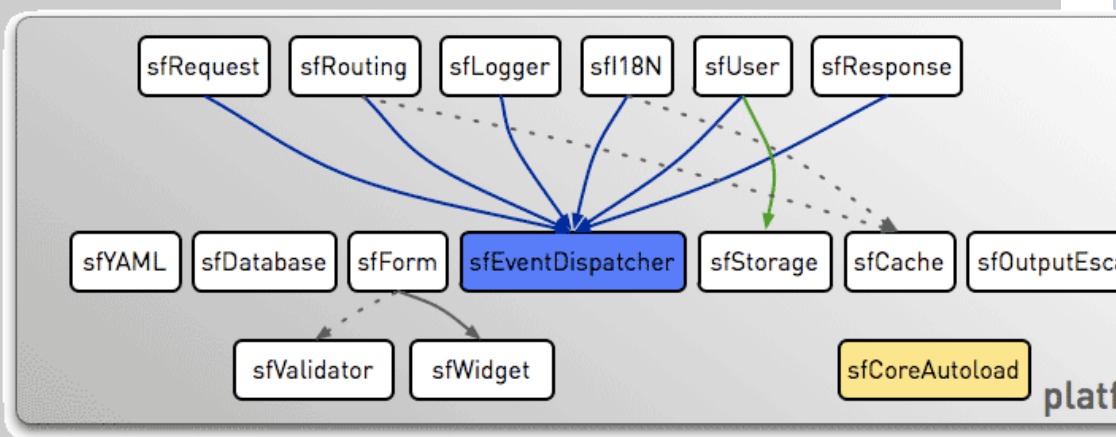


Utilisation du framework de formulaire sans symfony

Les composants du framework symfony sont assez découplés. Cela signifie que la plupart d'entre eux peuvent être utilisés sans utiliser l'ensemble du framework MVC. C'est le cas pour le framework de formulaire, qui n'a pas de dépendance sur symfony. Vous pouvez l'utiliser dans n'importe quelle application PHP en prenant les répertoires `lib/form/`, `lib/widgets/` et `lib/validators/`.

Un autre composant réutilisable est le framework de routage. Copiez le répertoire `lib/routing/` dans votre projet non-symfony, et profitez de jolies URL gratuitement.

Les composants qui sont indépendants de symfony forment la **plateforme de symfony**:



Soumission du formulaire

Ouvrons le fichier `jobActionsTest` pour ajouter des tests fonctionnels à la création d'emploi et au processus de validation.

A la fin du fichier, ajoutez le code suivant pour obtenir la page de création d'emploi :

```
// test/functional/frontend/jobActionsTest.php
$browser->info('3 - Post a Job page')->
  info('  3.1 - Submit a Job')->

  get('/job/new')->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'new')->
  end()
;
```

Nous avons déjà utilisé la méthode `click()` pour simuler des clics sur les liens. La même méthode `click()` peut être utilisée pour soumettre un formulaire. Pour un formulaire, vous

pouvez transmettre les valeurs à soumettre pour chaque champ en deuxième argument de la méthode. Comme un vrai navigateur, l'objet navigateur va fusionner les valeurs par défaut du formulaire avec les valeurs soumises.

Mais pour passer les valeurs des champs, nous avons besoin de connaître leurs noms. Si vous ouvrez le code source ou utilisez la fonctionnalité "Formulaires > Afficher les détails du formulaire" de la barre d'outils web de Firefox, vous verrez que le nom du champ company est `jobeet_job[company]`.

Lorsque PHP rencontre un champ de saisie avec un nom comme `jobeet_job[company]`, il le convertit automatiquement en un tableau nommé `jobeet_job`.

Pour rendre les choses un peu plus propre, nous allons changer le format de `job[%s]` en ajoutant le code suivant à la fin de la méthode `configure()` de `JobeetJobForm` :

```
// lib/form/doctrine/JobeetJobForm.class.php
$this->widgetSchema->setNameFormat('job[%s]');
```

Après ce changement, le nom de company devrait être `job[company]` dans votre navigateur. Il est maintenant temps de cliquer sur le bouton "Preview your job" et de passer les valeurs valides au formulaire :

```
// test/functional/frontend/jobActionsTest.php
$browser->info('3 - Post a Job page')->
  info('  3.1 - Submit a Job')->

  get('/job/new')->
  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'new')->
  end()->

  click('Preview your job', array('job' => array(
    'company'      => 'Sensio Labs',
    'url'          => 'http://www.sensio.com/',
    'logo'         => sfConfig::get('sf_upload_dir').'/jobs/sensio-labs.gif',
    'position'     => 'Developer',
    'location'     => 'Atlanta, USA',
    'description'  => 'You will work with symfony to develop websites for our
customers.',
    'how_to_apply' => 'Send me an email',
    'email'        => 'for.a.job@example.com',
    'is_public'    => false,
  )))->

  with('request')->begin()->
    isParameter('module', 'job')->
    isParameter('action', 'create')->
  end()
;
```

Le navigateur simule aussi des téléchargements de fichier, si vous passez le chemin absolu du fichier à télécharger.

Après la soumission du formulaire, nous avons vérifié que l'action exécutée est `create`.

Le testeur du formulaire

Le formulaire que nous avons soumis doit être valide. Vous pouvez tester cela en utilisant le **testeur de formulaire** :

```
with('form')->begin()->
  hasErrors(false)->
end()->
```

Le testeur de formulaire dispose de plusieurs méthodes pour tester l'état du formulaire actuel, comme les erreurs.

Si vous commettez une erreur lors de test, et que ce dernier ne passe pas, vous pouvez utiliser l'instruction `with('response')->~debug|Debug~()` que nous avons vu pendant le jour 9. Mais il vous faudra plonger dans le code HTML généré pour vérifier les messages d'erreur. Ce n'est pas vraiment commode. Le testeur de formulaire fournit également une méthode `debug()` qui

renvoie l'état du formulaire et de tous les messages d'erreur qui lui sont associés :

```
with('form')->debug()
```

Test de redirection

Comme le formulaire est valide, l'emploi aurait dû être créé et l'utilisateur redirigé vers la page `show` :

```
with('response')->isRedirected()->
followRedirect()->

with('request')->begin()->
  isParameter('module', 'job')->
  isParameter('action', 'show')->
end()->
```

Le `isRedirected()` teste si la page a été redirigée et la méthode `followRedirect()` suit la redirection.

La classe du navigateur ne suit pas les redirections automatiquement car vous voudrez peut-être introspecter les objets avant la redirection.

Le testeur de Doctrine

Finalement, nous voulons tester que l'emploi a été créé dans la base de données et vérifier que la colonne `is_activated` est mise à `false` car l'utilisateur ne l'a pas encore publié.

Cela peut se faire assez facilement en utilisant un autre testeur, le **testeur Doctrine**. Comme le testeur Doctrine n'est pas enregistré par défaut, ajoutons-le maintenant :

```
$browser->setTester('doctrine', 'sfTesterDoctrine');
```

Le testeur Doctrine fournit la méthode `check()` pour vérifier qu'un ou plusieurs objets dans la base de données correspondent à vos critères passée en argument.

```
with('doctrine')->begin()->
  check('JobeetJob', array(
    'location'     => 'Atlanta, USA',
    'is_activated' => false,
    'is_public'    => false,
  ))->
end()
```

Les critères peuvent être un tableau de valeurs comme ci-dessus, ou une instance `Doctrine_Query` pour les queries les plus complexes. Vous pouvez tester l'existence d'objets correspondants aux critères avec un booléen en troisième argument (la valeur par défaut est `true`), ou le nombre d'objets correspondants en passant un nombre entier.

Test pour les Erreurs

La création du formulaire d'emploi fonctionne comme prévu lorsque nous soumettons des valeurs valides. Ajoutons donc un test pour vérifier le comportement lorsque nous soumettons des données non valides :

```
$browser->
  info('  3.2 - Submit a Job with invalid values')->

  get('/job/new')->
  click('Preview your job', array('job' => array(
    'company'      => 'Sensio Labs',
    'position'     => 'Developer',
    'location'     => 'Atlanta, USA',
    'email'        => 'not.an.email',
  )))->

  with('form')->begin()->
    hasErrors(3)->
    isError('description', 'required')->
    isError('how_to_apply', 'required')->
    isError('email', 'invalid')->
  end()
```

```
;
```

La méthode `hasErrors()` permet de tester le nombre d'erreurs si un entier est passé. La méthode `isError()` teste le code erreur pour un champ donné.

Dans les tests que nous avons écrit pour la soumission de données non-valides , nous n'avons pas re-testé de nouveau le formulaire en entier. Nous avons seulement ajouté des tests pour des choses spécifiques.

Vous pouvez également tester le code HTML généré afin de vérifier qu'il contient les messages d'erreur, mais il n'est pas nécessaire dans notre cas car nous n'avons pas personnalisé la présentation de formulaire.

Maintenant, nous avons besoin de tester la barre d'administrateur trouvée sur la page de prévisualisation d'emploi. Lorsqu'un emploi n'a pas encore été activé, vous pouvez éditer, supprimer, ou publier l'emploi. Pour tester ces trois liens, nous aurons besoin de créer d'abord un emploi. Mais c'est beaucoup de copier et coller. Comme je n'aime pas gaspiller des e-arbres, nous allons ajouter une méthode de création d'emploi dans la classe `JobeetTestFunctional` :

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function createJob($values = array())
  {
    return $this->
      get('/job/new')->
      click('Preview your job', array('job' => array_merge(array(
        'company'      => 'Sensio Labs',
        'url'          => 'http://www.sensio.com/',
        'position'     => 'Developer',
        'location'     => 'Atlanta, USA',
        'description'  => 'You will work with symfony to develop websites for our
customers.',
        'how_to_apply' => 'Send me an email',
        'email'        => 'for.a.job@example.com',
        'is_public'    => false,
      ), $values)))->
      followRedirect()
    ;
  }

  // ...
}
```

La méthode `createJob()` crée un emploi, suit la redirection et retourne le navigateur pour ne pas casser la fluent interface. Vous pouvez également passer un tableau de valeurs qui sera fusionnée avec certaines valeurs par défaut.

Forcer la méthode HTTP d'un lien

Le test du lien "Publish" est maintenant plus simple :

```
$browser->info('  3.3 - On the preview page, you can publish the job')->
  createJob(array('position' => 'FOO1'))->
  click('Publish', array(), array('method' => 'put', '_with_csrf' => true))->

  with('doctrine')->begin()->
    check('JobeetJob', array(
      'position'     => 'FOO1',
      'is_activated' => true,
    ))->
  end()
;
```

Si vous vous souvenez du chapitre 10, le lien "Publish" a été configuré pour être appelé avec la méthode HTTP PUT. Comme les navigateurs ne comprennent pas les requêtes PUT, le helper `link_to()` convertit le lien vers un formulaire avec quelques JavaScript. Comme le navigateur de test n'exécute pas le code JavaScript, nous avons besoin de forcer la méthode à PUT en passant une troisième option à la méthode `click()`. En outre, le helper `link_to()` intégre aussi un jeton CSRF que nous avons activé par la protection CSRF durant le chapitre 1. L'option

`_with_csrf` simule ce jeton.

Tester le lien "Delete" est assez similaire :

```
$browser->info('  3.4 - On the preview page, you can delete the job')->
  createJob(array('position' => 'FOO2'))->
  click('Delete', array(), array('method' => 'delete', '_with_csrf' => true))->

  with('doctrine')->begin()->
    check('JobeetJob', array(
      'position' => 'FOO2',
    ), false)->
  end()
;
```

Les tests comme protection

Quand un emploi est publié, vous ne pouvez plus le modifier. Même si le lien "Edit" ne s'affiche plus sur la page de prévisualisation, ajoutons quelques tests de cette exigence.

D'abord, ajoutez un autre argument à la méthode `createJob()` pour permettre la publication automatique de l'emploi, et créez une méthode `getJobByPosition()` qui renvoie pour un emploi donné sa valeur pour `position` :

```
// lib/test/JobeetTestFunctional.class.php
class JobeetTestFunctional extends sfTestFunctional
{
  public function createJob($values = array(), $publish = false)
  {
    $this->
      get('/job/new')->
      click('Preview your job', array('job' => array_merge(array(
        'company'      => 'Sensio Labs',
        'url'          => 'http://www.sensio.com/',
        'position'     => 'Developer',
        'location'     => 'Atlanta, USA',
        'description'  => 'You will work with symfony to develop websites for our
customers.',
        'how_to_apply' => 'Send me an email',
        'email'        => 'for.a.job@example.com',
        'is_public'    => false,
      ), $values)))->
      followRedirect()
    ;

    if ($publish)
    {
      $this->
        click('Publish', array(), array('method' => 'put', '_with_csrf' => true))->
        followRedirect()
      ;
    }

    return $this;
  }

  public function getJobByPosition($position)
  {
    $q = Doctrine_Query::create()
      ->from('JobeetJob j')
      ->where('j.position = ?', $position);

    return $q->fetchOne();
  }

  // ...
}
```

Si un emploi est publié, la page d'édition doit retourner un code erreur 404 :

```
$browser->info('  3.5 - When a job is published, it cannot be edited anymore')->
  createJob(array('position' => 'FOO3'), true)->
  get(sprintf('/job/%s/edit', $browser->getJobByPosition('FOO3')->getToken()))->
```

```
  with('response')->begin()->
    isStatusCode(404)->
  end()
;
```

Mais si vous exécutez les tests, vous n'aurez pas le résultat escompté, car nous avons oublié hier d'implémenter cette mesure de sécurité. L'écriture des tests est aussi un excellent moyen de découvrir des bogues, car vous avez besoin de penser à tous les cas limites.

Résoudre le bogue est simple puisque nous avons juste besoin de transmettre une page 404 si l'emploi est activé :

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeEdit(sfWebRequest $request)
{
  $job = $this->getRoute()->getObject();
  $this->forward404If($job->getIsActivated());

  $this->form = new JobeetJobForm($job);
}
```

Le correctif est trivial, mais êtes-vous sûr que tout le reste fonctionne toujours comme prévu ? Vous pouvez ouvrir votre navigateur et commencer à tester toutes les combinaisons possibles pour accéder à la page d'édition. Mais il y a un moyen plus simple : lancez votre suite de tests, si vous avez introduit une régression, symfony va vous le dire tout de suite.

Retour vers le futur dans le test

Quand un emploi expire dans moins de cinq jours, ou s'il a déjà expiré, l'utilisateur peut étendre la validation de l'emploi pour 30 autres jours à compter de la date actuelle.

la date d'expiration est automatiquement réglé lorsque le travail est créé pour 30 jours dans le futur. Le test de cette exigence dans un navigateur n'est pas facile, car la date d'expiration est automatiquement définie dans 30 jours dans le futur lorsque l'emploi est créé. Ainsi, lors de la récupération de la page emploi, le lien n'est pas présent pour prolonger l'emploi. Bien sûr, vous pouvez adapter la date d'expiration dans la base de données, ou peaufiner le Template pour afficher en permanence le lien, mais c'est fastidieux et plutôt sensibles aux erreurs. Comme vous l'avez déjà deviné, l'écriture de tests nous aidera encore une fois.

Comme toujours, nous avons besoin d'ajouter une nouvelle route pour la première méthode `extend` :

```
# apps/frontend/config/routing.yml
job:
  class:   sfDoctrineRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: PUT, extend: PUT }
  requirements:
    token: \w+
```

Puis, mettez à jour le lien "Extend" codé dans le partial `_admin` :

```
<!-- apps/frontend/modules/job/templates/_admin.php -->
<?php if ($job->expiresSoon()): ?>
 - <?php echo link_to('Extend', 'job_extend', $job, array('method' => 'put')) ?> for
another <?php echo sfConfig::get('app_active_days') ?> days
<?php endif ?>
```

Puis, créez l'action `extend` :

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeExtend(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $this->forward404Unless($job->extend());

  $this->getUser()->setFlash('notice', sprintf('Your job validity has been extended
until %s.', $job->getDateTimeObject('expires_at')->format('m/d/Y')));
```

```
  $this->redirect('job_show_user', $job);
}
```

Comme prévu par l'action, la méthode `extend()` de `JobeetJob` retourne `true` si l'emploi a été prolongé ou `false` dans les autres cas :

```
// lib/model/doctrine/JobeetJob.class.php
class JobeetJob extends BaseJobeetJob
{
  public function extend()
  {
    if (!$this->expiresSoon())
    {
      return false;
    }

    $this->setExpiresAt(date('Y-m-d', time() + 86400 *
sfConfig::get('app_active_days')));

    $this->save();

    return true;
  }

  // ...
}
```

Enfin, ajoutez un scénario de test :

```
$browser->info('  3.6 - A job validity cannot be extended before the job expires
soon')->
  createJob(array('position' => 'FOO4'), true)->
  call(sprintf('/job/%s/extend', $browser->getJobByPosition('FOO4')->getToken()),
'put', array('_with_csrf' => true))->
  with('response')->begin()->
    isStatusCode(404)->
  end()
;

$browser->info('  3.7 - A job validity can be extended when the job expires soon')->
  createJob(array('position' => 'FOO5'), true)
;

$job = $browser->getJobByPosition('FOO5');
$job->setExpiresAt(date('Y-m-d'));
$job->save();

$browser->
  call(sprintf('/job/%s/extend', $job->getToken()), 'put', array('_with_csrf' =>
true))->
  with('response')->isRedirected()
;

$job->refresh();
$browser->test()->is(
  $job->getDateTimeObject('expires_at')->format('y/m/d'),
  date('y/m/d', time() + 86400 * sfConfig::get('app_active_days'))
);
```

Ce scénario de test introduit quelques nouveautés :

- La méthode `call()` récupère une URL avec une méthode différente GET ou POST
- Après que l'emploi a été mis à jour par l'action, nous avons besoin de recharger l'objet local avec `$job->refresh()`
- A la fin, nous utilisons l'objet incorporé `lime` directement pour tester la nouvelle date d'expiration.

Sécurité des formulaires

La magie de la sérialisation des formulaires !

Les formulaires Doctrine sont très faciles à utiliser car ils automatisent beaucoup de travail. Par exemple, la sérialisation d'un formulaire pour la base de données est aussi simple qu'un appel à `$form->save()`.

Mais comment ça marche ? Fondamentalement, la méthode `save()` suit les étapes suivantes:

- Commence une transaction (car les formulaires imbriqués Doctrine sont tous sauvés d'un seul coup)
- Processe les valeurs soumises (en appelant les méthodes `updateCOLUMNColumn()` si elles existent)
- Appelle la méthode `fromArray()` de l'objet Doctrine pour mettre à jour les valeurs de la colonne
- Sauve l'objet vers la base de données
- Commit la transaction

Fonctionnalités de sécurité intégrées

La méthode `fromArray()` prend un tableau de valeurs et met à jour les valeurs des colonnes correspondantes. Est-ce que ceci représente un problème de sécurité ? Que faire si quelqu'un essaie de sousmettre une valeur pour une colonne à laquelle il n'a pas l'autorisation ? Par exemple, puis-je forcer la colonne `token` ?

Commençons par écrire un test pour simuler une soumission d'un emploi avec un champ `token` :

```
// test/functional/frontend/jobActionsTest.php
$browser->
  get('/job/new')->
  click('Preview your job', array('job' => array(
    'token' => 'fake_token',
  )))->

  with('form')->begin()->
    hasErrors(7)->
    hasGlobalError('extra_fields')->
  end()
;
```

Lors de la soumission du formulaire, vous devez disposer d'une erreur globale `extra_fields`. C'est parce que les formulaires par défaut ne permettent pas que des champs supplémentaires soient présents dans les valeurs soumises. C'est aussi pourquoi tous les champs du formulaire doivent avoir un validateur associé.

Vous pouvez également soumettre des champs supplémentaires dans le confort de votre navigateur en utilisant des outils comme la barre d'outils Firefox pour les développeurs Web.

Vous pouvez contourner cette mesure de sécurité en mettant l'option `allow_extra_fields` à `true` :

```
class MyForm extends sfForm
{
  public function configure()
  {
    // ...

    $this->validatorSchema->setOption('allow_extra_fields', true);
  }
}
```

Le test doit maintenant passer mais la valeur `token` a été sortie des valeurs. Donc, vous n'êtes toujours pas en mesure de contourner la mesure de sécurité. Mais si vous voulez vraiment la valeur, définissez l'option `filter_extra_fields` à `false` :

```
$this->validatorSchema->setOption('filter_extra_fields', false);
```

Les tests écrits dans cette section sont uniquement à des fins de démonstration. Vous pouvez maintenant les retirer du projet Jobeet car les tests n'ont pas besoin de valider des caractéristiques de symfony.

Protection XSS et CSRF

Durant le chapitre 1, vous avez appris que la tâche `generate:app` crée une application sécurisée par défaut.

Premièrement, elle a activé la protection contre les attaques XSS. Cela signifie que toutes les variables utilisées dans les Templates sont échappés par défaut. Si vous essayez de soumettre une description d'un emploi avec certaines balises HTML à l'intérieur, vous remarquerez que lorsque symfony rend la page des emplois, les balises HTML de la description ne sont pas interprétées, mais rendues en texte brut.

Puis, elle a permis la protection CSRF. Quand un jeton CSRF est fixé, tous les formulaires intégrent un champ caché `_csrf_token`.

La stratégie d'échapement et le secret CSRF peuvent être modifiés à tout moment en modifiant le fichier de configuration `apps/frontend/config/settings.yml`. Comme pour le fichier `databases.yml`, les paramètres sont configurables par environnement :

```
all:
  .settings:
    # Form security secret (CSRF protection)
    csrf_secret: Unique$ecret

    # Output escaping settings
    escaping_strategy: true
    escaping_method:   ESC_SPECIALCHARS
```

Les tâches de maintenance

Même si symfony est un framework web, il est livré avec un outil de ligne de commande|Ligne de commande. Vous l'avez déjà utilisé pour créer la structure des répertoires par défaut du projet et de l'application, mais aussi pour générer différents fichiers pour le modèle. L'ajout d'une nouvelle tâche est plutôt facile car les outils utilisés par la ligne de commande de symfony sont intégrés dans un framework.

Lorsqu'un utilisateur crée un emploi, il doit l'activer pour le mettre en ligne. Mais sinon, la base de données grandira avec des vieux emplois. Nous allons créer une tâche qui suppriment de vieux emplois de la base de données. Cette tâche devra être exécuté régulièrement dans une tâche cron.

```
// lib/task/JobeetCleanupTask.class.php
class JobeetCleanupTask extends sfBaseTask
{
  protected function configure()
  {
    $this->addOptions(array(
      new sfCommandOption('application', null, sfCommandOption::PARAMETER_REQUIRED,
'The application', 'frontend'),
      new sfCommandOption('env', null, sfCommandOption::PARAMETER_REQUIRED, 'The
environement', 'prod'),
      new sfCommandOption('days', null, sfCommandOption::PARAMETER_REQUIRED, '', 90),
    ));

    $this->namespace = 'jobeet';
    $this->name = 'cleanup';
    $this->briefDescription = 'Cleanup Jobeet database';

    $this->detailedDescription = <<<EOF
The [jobeet:cleanup|INFO] task cleans up the Jobeet database:

  [./symfony jobeet:cleanup --env=prod --days=90|INFO]
EOF;
  }

  protected function execute($arguments = array(), $options = array())
  {
    $databaseManager = new sfDatabaseManager($this->configuration);

    $nb = Doctrine_Core::getTable('JobeetJob')->cleanup($options['days']);
    $this->logSection('doctrine', sprintf('Removed %d stale jobs', $nb));
  }
```

```
}
```

La configuration des tâches se fait dans la méthode `configure()`. Chaque tâche doit avoir un nom unique (`namespace:name`), et peut avoir des arguments et des options.

Parcourez les tâches intégrées de symfony (`lib/task/`) pour plus d'exemples d'utilisation.

La tâche `jobeet:cleanup` définit deux options : `--env` et `--days` avec certaines valeurs par défaut raisonnables.

L'exécution de la tâche est similaire à l'exécution des autres tâches intégrées de symfony :

```
$ php symfony jobeet:cleanup --days=10 --env=dev
```

Comme toujours, le code de nettoyage de la base de données a été pris dans la classe `JobeetJobTable` :

```
// lib/model/doctrine/JobeetJobTable.class.php
public function cleanup($days)
{
  $q = $this->createQuery('a')
    ->delete()
    ->andWhere('a.is_activated = ?', 0)
    ->andWhere('a.created_at < ?', date('Y-m-d', time() - 86400 * $days));

  return $q->execute();
}
```

Les tâches symfony se comportent bien avec leur environnement, car ils renvoient une valeur en fonction du succès de la tâche. Vous pouvez forcer une valeur de retour en retournant un entier de manière explicite à la fin de la tâche.

Conclusion

Les tests sont au cœur de la philosophie et des outils de symfony . Dans ce chapitre, nous avons encore appris comment exploiter les outils symfony pour rendre le processus de développement plus facile, plus rapide et plus important, plus sûr.

Le framework de formulaire symfony offre beaucoup plus que de simples widgets et validateurs : il vous donne un moyen simple de tester vos formulaires et assurer que vos formulaires sont sécurisés par défaut.

Notre tour des fonctionnalités de symfony ne s'achève pas aujourd'hui. Dans le prochain chapitre, nous allons créer l'application backend pour Jobeet. La création d'une interface backend est un must pour la plupart des projets web et Jobeet n'est pas différent. Mais comment allons-nous pouvoir développer une telle interface en seulement une heure ? Simple, nous allons utiliser le framework admin generator de symfony. Jusque-là, prenez garde.

« Jour 10 : Les formulaires

Jour 12 : L'Admin Generator »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

development, training, consulting.
Sensio Labs also supports several large Open-Source projects.

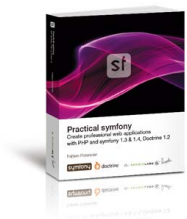
Swift Mailer - Mailing library
Twig - Template library
Pirum - PEAR channel server

Books -
Confere
Confere

Practical symfony
Jour 12 : L'Admin Generator

Avec les ajouts que nous avons faits hier sur Jobeet, l'application frontend est maintenant pleinement utilisable par les chercheurs d'emploi et les posteurs d'emplois. Il est temps de parler un peu de l'application backend.

Dans ce chapitre, grâce à la fonctionnalité de l'admin generator de symfony, nous allons développer une interface backend complète pour Jobeet en seulement une heure.



La création du backend

La toute première étape consiste à créer l'application backend. Si vous vous souvenez bien, vous devriez vous rappeler comment le faire avec la tâche `generate:app` :

```
$ php symfony generate:app backend
```

L'application backend est maintenant disponible à l'adresse `http://www.jobeet.com.localhost/backend.php/` pour l'environnement de prod et `http://www.jobeet.com.localhost/backend_dev.php/` pour l'environnement de dev.

Lorsque vous avez créé l'application frontend, le contrôleur frontal de la production a été nommé `index.php`. Comme vous ne pouvez avoir qu'un seul fichier `index.php` par répertoire, symfony crée un fichier `index.php` pour le premier contrôleur de la production et les noms des autres après le nom d'application.

Si vous essayez de recharger les données de tests avec la tâche `doctrine:data-load`, elle ne fonctionnera plus. C'est parce que la méthode `JobeetJob::save()` a besoin d'accéder au fichier de configuration `app.yml` depuis l'application frontend. Comme nous avons maintenant deux applications, symfony utilise le premier qu'il trouve, qui est maintenant le backend.

Mais comme on le voit pendant le jour 8, les paramètres peuvent être configurés à différents niveaux. En déplaçant le contenu du fichier `apps/frontend/config/app.yml` vers `config/app.yml`, les paramètres seront partagés par toutes les applications et le problème sera réglé. Changez le maintenant car nous allons utiliser les classes du modèle assez largement dans l'admin generator, et donc nous aurons besoin des variables définies dans `app.yml` dans l'application backend.

La tâche `doctrine:data-load` prend également en charge une option `--application`. Donc, si vous avez besoin de paramètres spécifiques d'une application ou une autre, c'est la marche à suivre :

```
$ php symfony doctrine:data-load --application=frontend
```

Les modules du backend

Pour l'application frontend, la tâche `doctrine:generate-module` a été utilisé pour démarrer un module de base CRUD basé sur une classe modèle. Pour le backend, la tâche `doctrine:generate-admin` sera utilisé car il génère une interface de travail complète de backend pour une classe de modèle :

```
$ php symfony doctrine:generate-admin backend JobeetJob --module=job
$ php symfony doctrine:generate-admin backend JobeetCategory --module=category
```

Ces deux commandes créent un module `job` et un module `category` pour les classes respectives du modèle `JobeetJob` et `JobeetCategory`.

L'option facultative `--module` remplace le nom module généré par défaut par la tâche (ce qui

aurait été autrement `jobeet_job` pour la classe `JobeetJob`).

Derrière le rideau, la tâche a également créé une route personnalisée pour chaque module :

```
# apps/backend/config/routing.yml
jobeet_job:
  class: sfDoctrineRouteCollection
  options:
    model:                JobeetJob
    module:               job
    prefix_path:          job
    column:               id
    with_wildcard_routes: true
```

Il ne faut donc pas s'étonner que la classe de la route utilisée par l'admin generator|Admin Generator est `sfDoctrineRouteCollection`, car l'objectif principal d'une interface d'administration est la gestion du cycle de vie des objets du modèle.

La définition de la route définit également certaines options que nous n'avons pas vu auparavant :

- `prefix_path`: Définit le chemin du préfixe pour la route générée (par exemple, la page d'édition sera quelque chose comme `/job/1/edit`).
- `column`: Définit la colonne de table à utiliser dans l'URL pour les liens faisant référence à un objet.
- `with_wildcard_routes`: Comme l'interface d'administration aura plus que des opérations classiques du CRUD, cette option permet de définir plus d'objet et une collection d'actions sans modifier la route.

Comme toujours, c'est une bonne idée de lire l'aide avant d'utiliser une nouvelle tâche.

```
$ php symfony help doctrine:generate-admin
```

Elle vous donnera tous les arguments de la tâche et les options ainsi que quelques exemples d'utilisation classique.

Backend Look and Feel

D'emblée, vous pouvez utiliser les modules générés :

```
http://www.jobeet.com.localhost/backend_dev.php/job
http://www.jobeet.com.localhost/backend_dev.php/category
```

Les modules admin ont beaucoup plus de fonctionnalités que les modules simples que nous avons générés les jours précédents. Sans écrire une seule ligne de PHP, chaque module fournit ces fonctionnalités exceptionnelles :

- La liste des objets est **paginée**
- La liste est **triable**
- La liste peut être **filtrée**
- Les objets peuvent être **créés**, **édités** et **supprimés**
- Les objets sélectionnés peuvent être supprimés dans un **batch**
- La **validation** du formulaire est activée
- Les **messages flash** donne immédiatement un retour à l'utilisateur
- ... et plus encore

L'admin generator offre toutes les fonctions dont vous avez besoin pour créer une interface backend dans un simple package à configurer.

Si vous tentez de consulter les deux modules d'administration générés, vous remarquerez qu'aucun style graphique n'est activé pour l'instant alors que le générateur d'administration de symfony en fournit un par défaut. Il ne s'agit ni d'un bug ni d'un oubli de la part du développeur. En réalité, les ressources web du plugin `sfDoctrinePlugin` ne sont pas encore publiées dans le dossier `web/`. Nous devons donc publier ces ressources web dans le répertoire `web/` du projet à l'aide de la commande `plugin:publish-assets`:

```
$ php symfony plugin:publish-assets
```

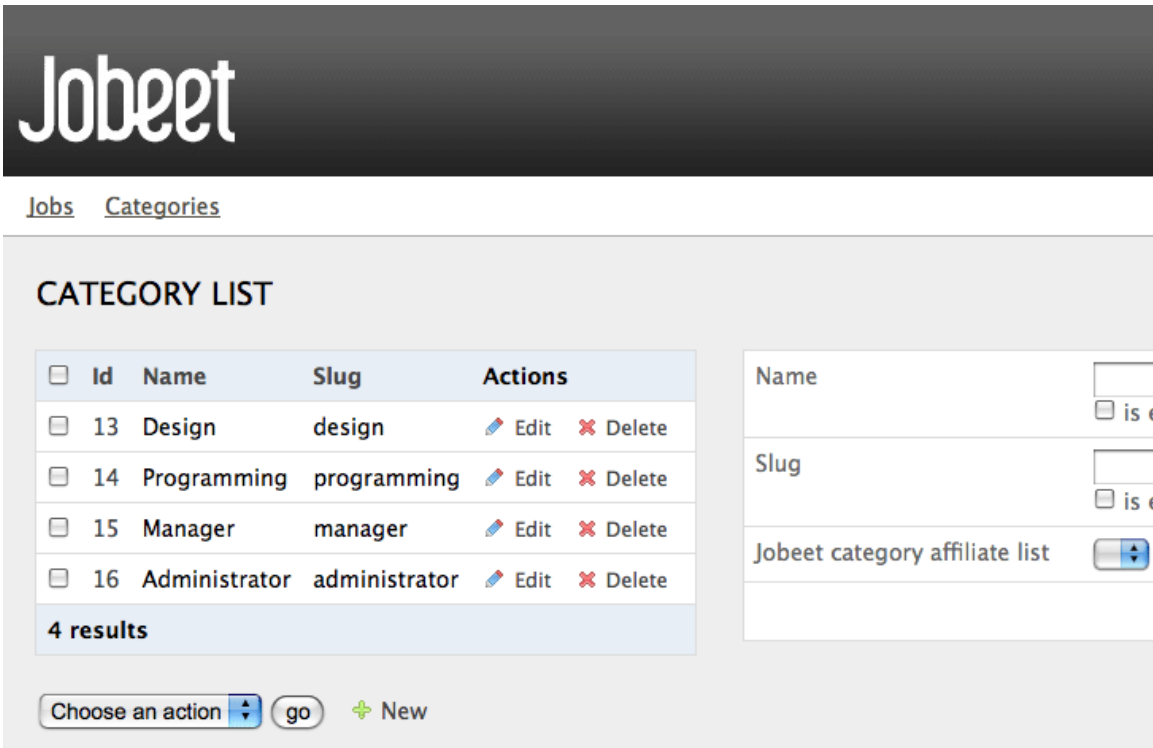
Pour rendre l'expérience utilisateur un peu mieux, nous avons besoin de personnaliser le backend par défaut. Nous allons aussi ajouter un menu simple pour le rendre facile pour

naviguer entre les différents modules.

Remplacez le contenu du fichier `~layout|Layout~.php` par défaut avec le code ci-dessous :

```
// apps/backend/templates/layout.php
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Jobeet Admin Interface</title>
    <link rel="shortcut icon" href="/favicon.ico" />
    <?php use_stylesheet('admin.css') ?>
    <?php include_javascripts() ?>
    <?php include_stylesheets() ?>
  </head>
  <body>
    <div id="container">
      <div id="header">
        <h1>
          <a href="<?php echo url_for('homepage') ?>">
            <img src="/images/logo.jpg" alt="Jobeet Job Board" />
          </a>
        </h1>
      </div>

      <div id="menu">
        <ul>
          <li>
            <?php echo link_to('Jobs', 'jobeet_job') ?>
          </li>
          <li>
            <?php echo link_to('Categories', 'jobeet_category') ?>
          </li>
        </ul>
      </div>

      <div id="content">
        <?php echo $sf_content ?>
      </div>

      <div id="footer">
        <img src="/images/jobeet-mini.png" />
        powered by <a href="http://www.symfony-project.org/">
        <img src="/images/symfony.gif" alt="symfony framework" /></a>
      </div>
    </div>
  </body>
</html>
```

Cette mise en page utilise une feuille de style `admin.css`. Ce fichier doit être déjà présent dans `web/css/` car il a été installé avec les autres feuilles de style au cours du chapitre 4.



Éventuellement, changez la page d'accueil par défaut de symfony dans `routing.yml` :

```
# apps/backend/config/routing.yml
homepage:
  url:   /
  param: { module: job, action: index }
```

Le cache de symfony

Si vous êtes curieux, vous avez probablement déjà ouvert les fichiers générés par la tâche sous le répertoire `apps/backend/modules/`. Sinon, s'il vous plaît ouvrez-les maintenant. Surprise ! Les répertoires `templates` sont vides et les fichiers `actions.class.php` sont assez vides :

```
// apps/backend/modules/job/actions/actions.class.php
require_once dirname(__FILE__).'/../lib/jobGeneratorConfiguration.class.php';
require_once dirname(__FILE__).'/../lib/jobGeneratorHelper.class.php';

class jobActions extends autoJobActions
{
}
```

Comment peut-il fonctionner ? Si vous regardez d'un peu plus près, vous remarquerez que la classe `jobActions` étend `autoJobActions`. La classe `autoJobActions` est automatiquement généré par symfony si elle n'existe pas. Elle se trouve dans le répertoire `cache/backend/dev/modules/autoJob/`, qui contient le module «réel» :

```
// cache/backend/dev/modules/autoJob/actions/actions.class.php
class autoJobActions extends sfActions
{
  public function preExecute()
  {
    $this->configuration = new jobGeneratorConfiguration();

    if (!$this->getUser()->hasCredential(
      $this->configuration->getCredentials($this->getActionName())
    ))
    {

// ...
```

La manière dont l'admin generator fonctionne devrait vous rappeler certains comportements connus. En fait, il est assez semblable à ce que nous avons appris sur le modèle et les classes du formulaire. Basé sur la définition du schéma du modèle, symfony génère le modèle et les

classes du formulaire. Pour l'admin generator, le module généré peut être configuré en éditant le fichier `config/generator.yml` trouvé dans le module :

```
# apps/backend/modules/job/config/generator.yml
generator:
  class: sfDoctrineGenerator
  param:
    model_class:           JobeetJob
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          jobeet_job
    with_doctrine_route:   true

    config:
      actions: ~
      fields:  ~
      list:    ~
      filter:  ~
      form:    ~
      edit:    ~
      new:     ~
```

Chaque fois que vous mettez à jour le fichier `generator.yml`, symfony régénère le cache. Comme nous le voyons aujourd'hui, la personnalisation de l'admin des modules générés est simple, rapide et amusant.

La re-génération automatique des fichiers de cache se produit uniquement dans l'environnement de développement. En production, vous devez effacer le cache manuellement avec la tâche `cache:clear`.

La configuration du backend

Un module admin peut être personnalisé en modifiant la clé `config` du fichier `generator.yml`. La configuration est organisée en sept sections :

- `actions`: La configuration par défaut pour les actions figurant sur la liste et dans les formulaires
- `fields`: La configuration par défaut pour les champs
- `list`: La configuration pour la liste
- `filter`: La configuration pour les filtres
- `form`: La configuration pour le formulaire new/edit
- `edit`: La configuration spécifique pour la page edit
- `new`: La configuration spécifique pour la page new

Commençons la personnalisation.

Configuration du titre

Les titres des sections `list`, `edit` et `new` du module `category` peuvent être personnalisés en définissant l'option `title` :

```
# apps/backend/modules/category/config/generator.yml
config:
  actions: ~
  fields:  ~
  list:
    title: Category Management
  filter:  ~
  form:    ~
  edit:
    title: Editing Category "%%name%%"
  new:
    title: New Category
```

Le `title` pour la section `edit` contient des valeurs dynamiques : toutes les chaînes entourées par `%%` sont remplacées par les valeurs de la colonne pour l'objet correspondant.

La configuration pour le module `job` est assez similaire :

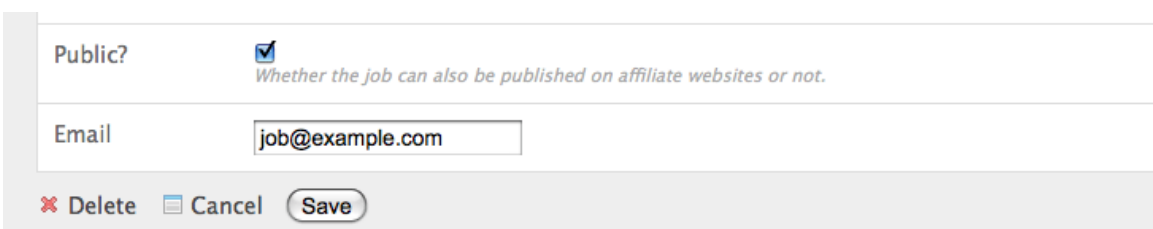
```
# apps/backend/modules/job/config/generator.yml
config:
  actions: ~
  fields:  ~
  list:
    title: Job Management
  filter:  ~
  form:    ~
  edit:
    title: Editing Job "%%company%% is looking for a %%position%%"
  new:
    title: Job Creation
```

Configuration des champs

Les différentes vues (`list`, `new` et `edit`) sont composés de champs. Un champ peut être une colonne de la classe du modèle, ou une colonne virtuelle comme nous le verrons plus tard.

La configuration des champs par défaut peut être personnalisée avec la section `fields` :

```
# apps/backend/modules/job/config/generator.yml
config:
  fields:
    is_activated: { label: Activated?, help: Whether the user has activated the job,
or not }
    is_public:    { label: Public?, help: Whether the job can also be published on
affiliate websites, or not }
```



La section `fields` remplace la configuration des champs pour toutes les vues, ce qui signifie que le `label` pour le champ `is_activated` sera changé pour les vues `list`, `edit` et `new`.

La configuration de l'admin generator est basé sur un principe de configuration en cascade. Par exemple, si vous souhaitez modifier un label pour la vue `list` uniquement, définissez une option `fields` sous la section `list` :

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    fields:
      is_public:    { label: "Public? (label for the list)" }
```

Toute configuration qui est définie dans la section principale `fields` peut être substituée par la configuration de la vue spécifique. Les règles de substitution sont les suivantes :

- `new` et `edit` héritent de `form` qui hérite `fields`
- `list` hérite `fields`

`filter` hérite `fields`

Pour les sections de formulaire (`form`, `edit` et `new`), les options `label` et `help` substituent celles définies dans les classes de formulaires.

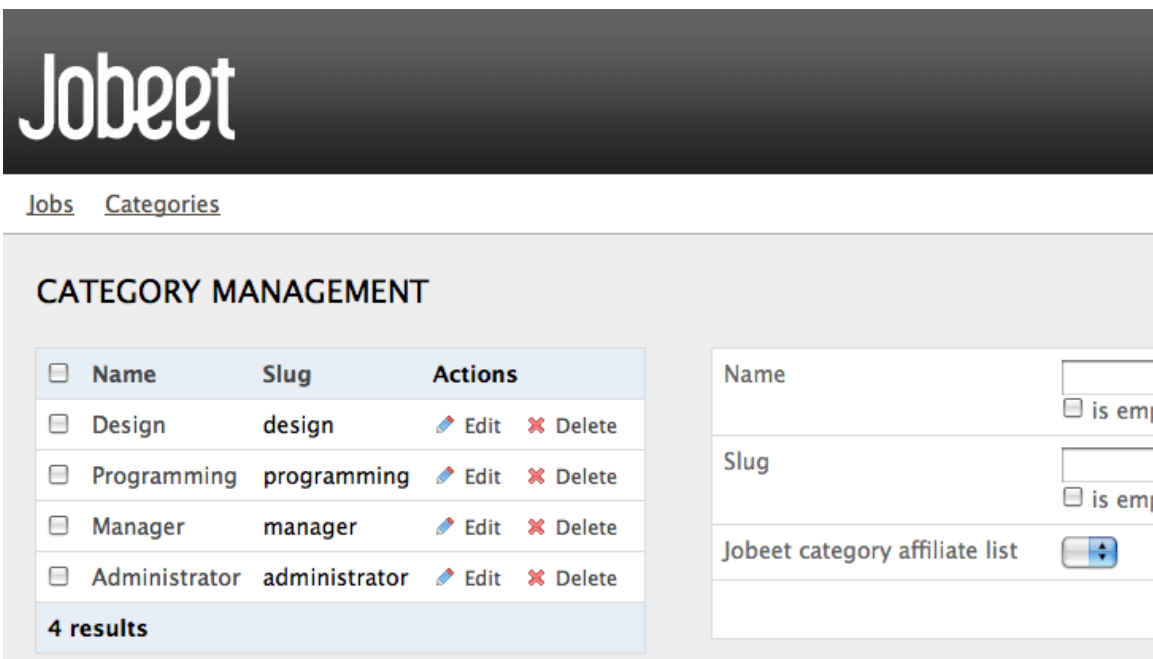
Configuration de la vue list

display

Par défaut, les colonnes de la vue de la liste sont toutes des colonnes du modèle, dans l'ordre du fichier du schéma. L'option `display` substitue l'ordre par défaut des colonnes à afficher :

```
# apps/backend/modules/category/config/generator.yml
config:
  list:
    title:   Category Management
    display: [=name, slug]
```

Le signe = avant la colonne `name` est une convention visant à convertir la chaîne en un lien.



Faisons de même pour le module `job` pour le rendre plus lisible :

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    title:   Job Management
    display: [company, position, location, url, is_activated, email]
```

layout

La liste peut être affichée avec des layouts différents. Par défaut, le layout est `tabular`, ce qui signifie que chaque valeur de colonne est la colonne dans sa propre table. Mais pour le module `job`, il serait préférable d'utiliser le layout `stacked`, qui est l'autre layout intégré :

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    title:   Job Management
    layout:  stacked
    display: [company, position, location, url, is_activated, email]
    params:  |
      %%is_activated%% <small>%%category_id%%</small> - %%company%%
       (<em>%%email%%</em>) is looking for a %%=position%% (%%location%%)
```

Dans un layout `stacked`, chaque objet est représenté par une seule chaîne, qui est definie par l'option `params`.

L'option `display` est toujours nécessaire, car elle définit les colonnes qui seront triables par l'utilisateur.

Les colonnes "Virtuelles"

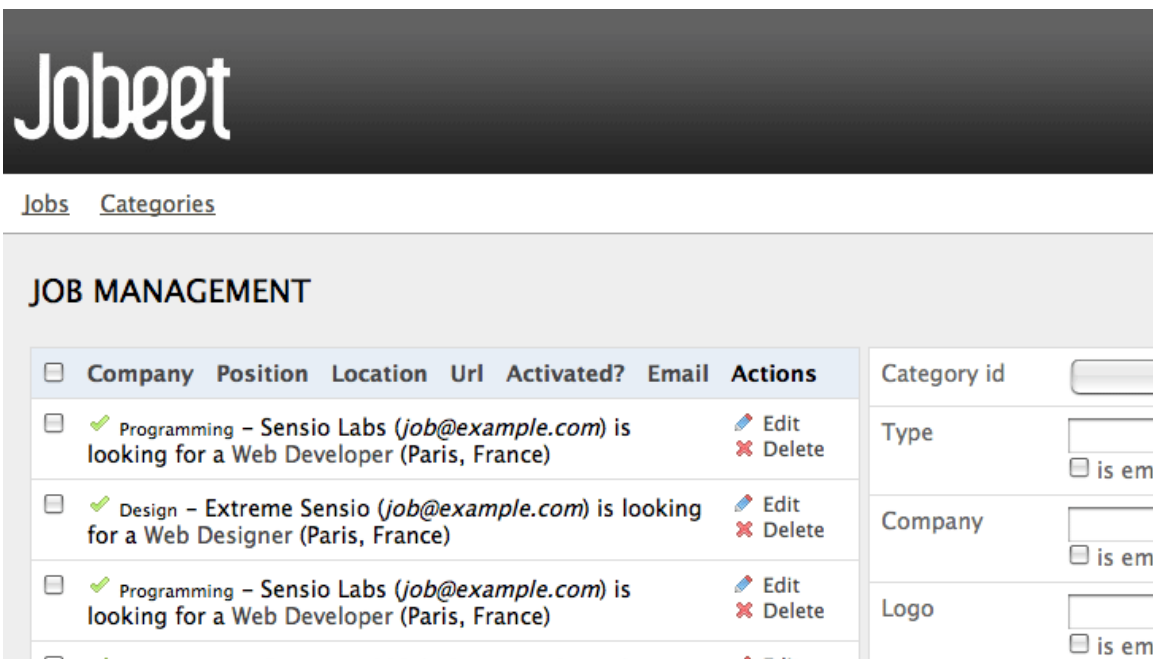
Avec cette configuration, le segment `%%category_id%%` sera remplacée par la clé primaire de la catégorie. Mais il serait plus utile d'afficher le nom de la catégorie.

Chaque fois que vous utilisez la notation `%%`, la variable ne doit pas nécessairement correspondre à une colonne réelle dans le schéma de la base de données. L'admin generator a seulement besoin de trouver un getter associé dans la classe du modèle.

Pour afficher le nom de la catégorie, on peut définir une méthode `getCategoryName()` dans la classe du modèle `JobeetJob` et remplacer `%%category_id%%` par `%%category_name%%`.

Mais la classe `JobeetJob` a déjà une méthode `getJobeetCategory()` qui retourne l'objet catégorie associé. Et si vous utilisez `%%jobeet_category%%`, cela fonctionne comme la classe `JobeetCategory`. Cette dernière a une méthode magique `__toString()` qui convertit l'objet en une chaîne.

```
# apps/backend/modules/job/config/generator.yml
%%is_activated%% <small>%%jobeet_category%%</small> - %%company%%
 (<em>%%email%%</em>) is looking for a %%=position%% (%%location%%)
```



sort

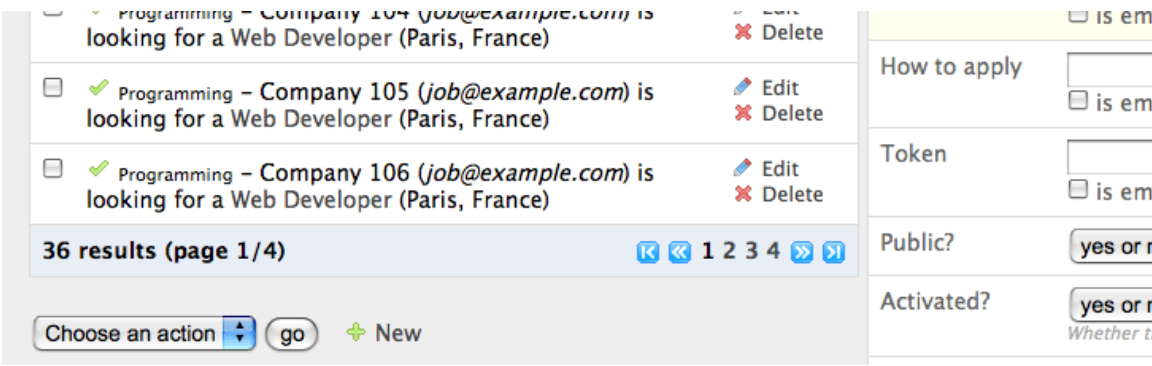
En tant qu'administrateur, vous serez probablement plus intéressés de voir les dernières offres d'emploi postées. Vous pouvez configurer la colonne de tri par défaut en ajoutant l'option `sort` :

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    sort: [expires_at, desc]
```

max_per_page

Par défaut, la liste est paginée et chaque page contient 20 articles. Ceci peut être modifié avec l'option `max_per_page` :

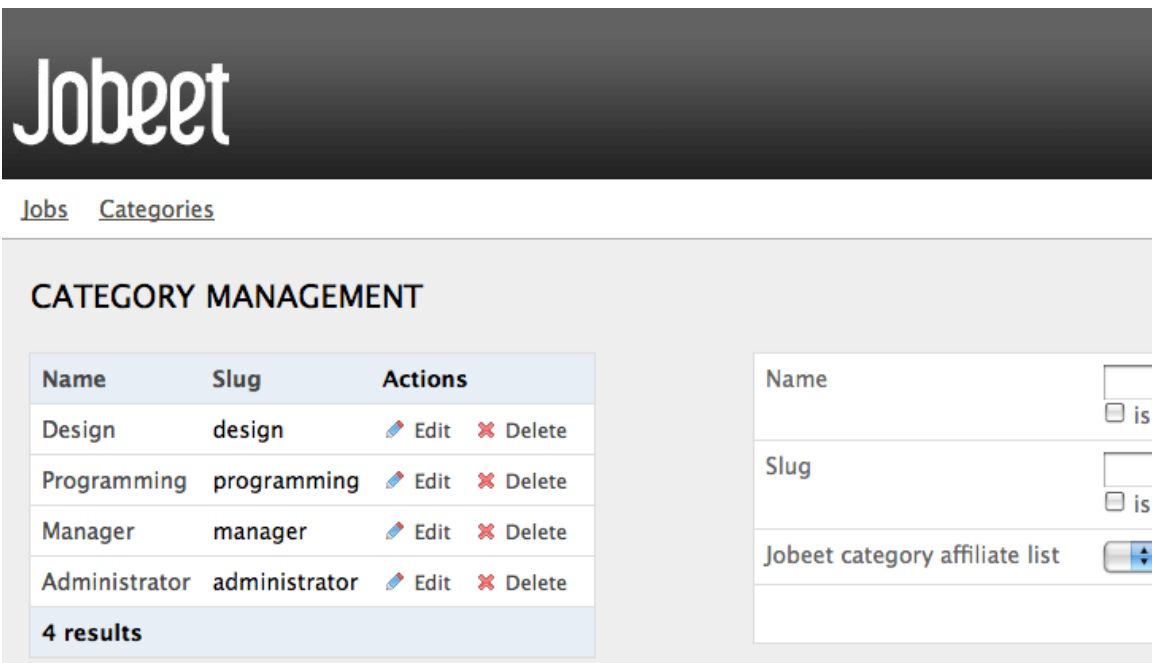
```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    max_per_page: 10
```



batch_actions

Sur une liste, une action peut être exécutée sur plusieurs objets. Ces actions batch ne sont pas nécessaires pour le module `category`, donc nous allons les supprimer :

```
# apps/backend/modules/category/config/generator.yml
config:
  list:
    batch_actions: {}
```



L'option `batch_actions` définit la liste des actions batch. Le tableau vide permet l'élimination de la fonctionnalité.

Par défaut, chaque module a une action batch `delete` définie par le framework, mais pour le module `job`, supposons que nous devons trouver un moyen de prolonger la validité de certains emplois sélectionnés pour 30 jours supplémentaires :

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    batch_actions:
      _delete:    ~
      extend:     ~
```

Toutes les actions commençant par un `_` sont intégrés dans les actions fournies par le framework. Si vous actualisez votre navigateur et sélectionnez les actions batch supplémenaires, symfony va lever une exception en vous invitant à créer une méthode `executeBatchExtend()` :

```
// apps/backend/modules/job/actions/actions.class.php
class jobActions extends autoJobActions
{
  public function executeBatchExtend(sfWebRequest $request)
  {
```

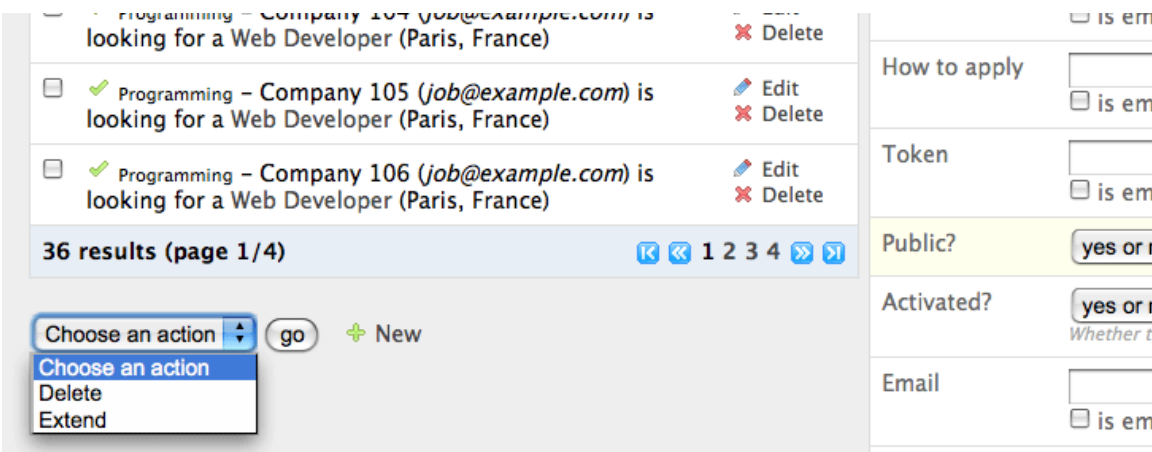
```
    $ids = $request->getParameter('ids');

    $q = Doctrine_Query::create()
      ->from('JobeetJob j')
      ->whereIn('j.id', $ids);

    foreach ($q->execute() as $job)
    {
      $job->extend(true);
    }

    $this->getUser()->setFlash('notice', 'The selected jobs have been extended
successfully.');

    $this->redirect('jobeet_job');
  }
}
```

Les clés primaires sélectionnées sont stockées dans le paramètre de requête `ids`. Pour chaque emploi sélectionné, la méthode `JobeetJob::extend()` est appelée avec un argument supplémentaire pour contourner le contrôle d'expiration.

Mettez à jour la méthode `extend()` pour prendre ce nouvel argument en compte :

```
// lib/model/doctrine/JobeetJob.class.php
class JobeetJob extends BaseJobeetJob
{
  public function extend($force = false)
  {
    if (!$force && !$this->expiresSoon())
    {
      return false;
    }

    $this->setExpiresAt(date('Y-m-d', time() + 86400 *
sfConfig::get('app_active_days')));
    $this->save();

    return true;
  }

  // ...
}
```

Après que tous les emplois aient été étendus, l'utilisateur est redirigé vers la page d'accueil du module `job`.



object_actions

Dans la liste, il y a une colonne supplémentaire pour les actions que vous pouvez exécuter sur un seul objet. Pour le module `category`, retirons les car nous avons un lien sur le nom de catégorie pour la modifier, et nous n'avons pas vraiment besoin d'être en mesure d'en supprimer une directement dans la liste :

```
# apps/backend/modules/category/config/generator.yml
```

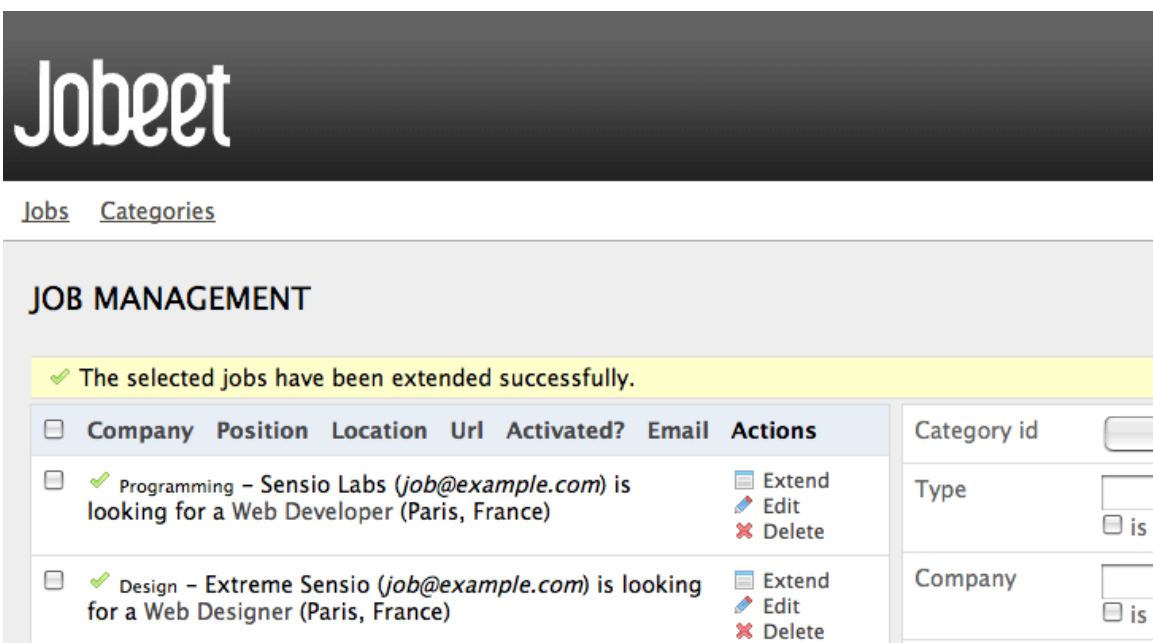
```
config:
  list:
    object_actions: {}
```

Pour le module `job`, gardons les actions existantes et ajoutons un nouvel `extend` d'une action semblable à celle que nous avons ajouté pour les actons batch :

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    object_actions:
      extend:     ~
      _edit:      ~
      _delete:    ~
```

Quant aux actions batch, les actions `_delete` et `_edit` sont celles définies par le framework. Nous avons besoin de définir l'action `listExtend()` pour faire fonctionner le lien `extend` :

```
// apps/backend/modules/job/actions/actions.class.php
class jobActions extends autoJobActions
{
  public function executeListExtend(sfWebRequest $request)
  {
    $job = $this->getRoute()->getObject();
    $job->extend(true);

    $this->getUser()->setFlash('notice', 'The selected jobs have been extended
successfully.');

    $this->redirect('jobeet_job');
  }

  // ...
}
```



actions

Nous avons déjà vu comment lier une action à une liste d'objets ou à un objet unique. L'option `actions` définit les actions qui ne prennent pas d'objet du tout, comme la création d'un nouvel objet. Enlevons l'action par défaut `new` et ajoutons une nouvelle action, qui supprime tous les emplois qui n'ont pas été activés par l'employeur pendant plus de 60 jours :

```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    actions:
      deleteNeverActivated: { label: Delete never activated jobs }
```

Jusqu'à présent, toutes les actions que nous avons définies ont ~, ce qui signifie que symfony configure automatiquement l'action. Chaque action peut être personnalisée en définissant un tableau de paramètres. L'option `label` substitue le label par défaut généré par symfony.

Par défaut, l'action exécutée lorsque vous cliquez sur le lien est le nom de l'action préfixée par `list`.

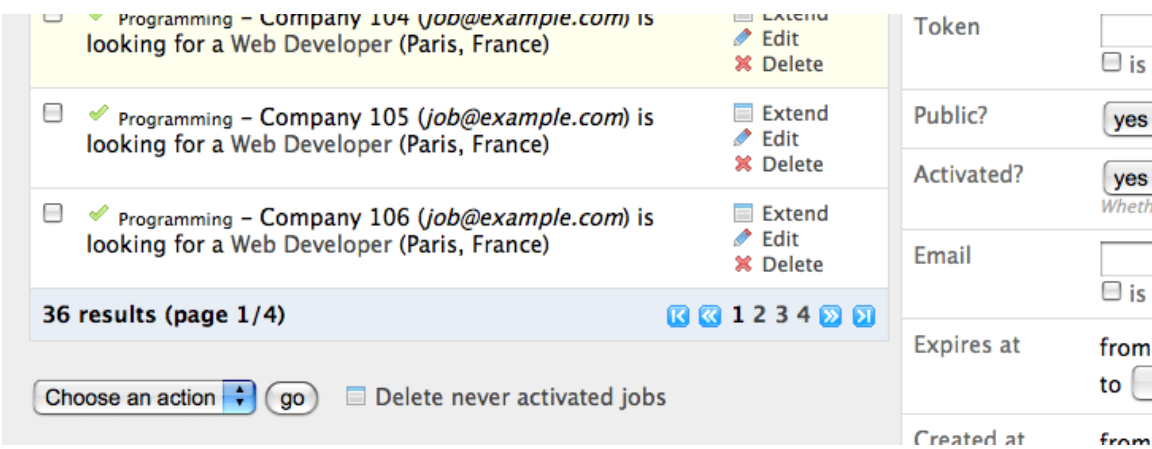
Créez l'action `listDeleteNeverActivated` dans le module `job` :

```
// apps/backend/modules/job/actions/actions.class.php
class jobActions extends autoJobActions
{
  public function executeListDeleteNeverActivated(sfWebRequest $request)
  {
    $nb = Doctrine::getTable('JobeetJob')->cleanup(60);

    if ($nb)
    {
      $this->getUser()->setFlash('notice', sprintf('%d never activated jobs have been deleted successfully.', $nb));
    }
    else
    {
      $this->getUser()->setFlash('notice', 'No job to delete.');
    }

    $this->redirect('jobeet_job');
  }

  // ...
}
```

Nous avons réutilisé la méthode `JobeetJobTable::cleanup()` définie dans le précédent chapitre. C'est un autre excellent exemple de la réutilisation fournies par le modèle MVC.

Vous pouvez également modifier l'action à exécuter en passant le paramètre `action` :

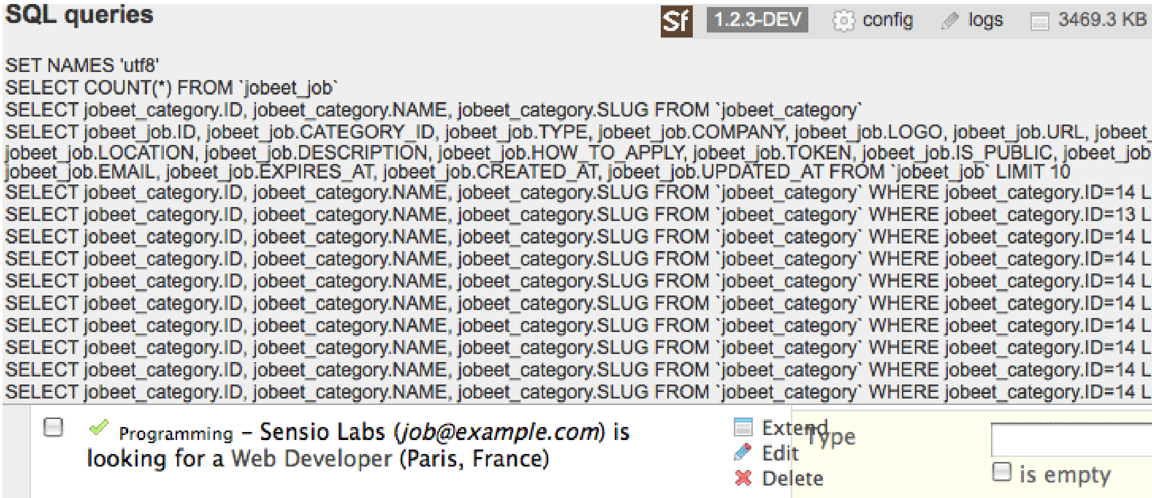
```
deleteNeverActivated: { label: Delete never activated jobs, action: foo }
```



table_method

Le nombre de requêtes nécessaires vers la base de données pour afficher la page de la liste des emplois est de 14, comme illustré par le web debug toolbar.

Si vous cliquez sur le nombre, vous verrez que la plupart des requêtes sont utilisées pour récupérer le nom de la catégorie pour chaque emploi :



Pour réduire le nombre de requête, nous pouvons changer la méthode par défaut utilisé pour récupérer les emplois en utilisant l'option `table_method` :

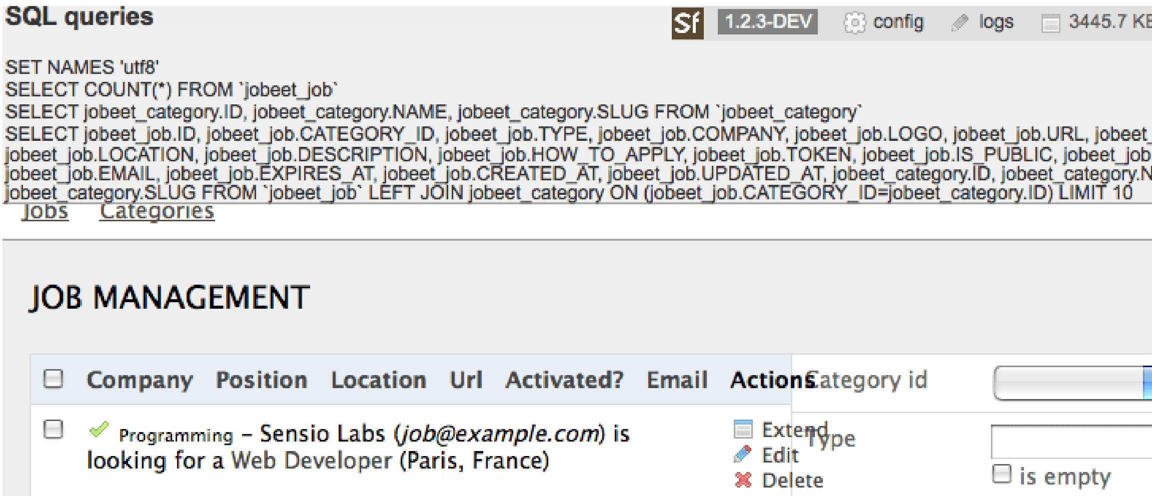
```
# apps/backend/modules/job/config/generator.yml
config:
  list:
    table_method: retrieveBackendJobList
```

Maintenant vous devez créer la méthode `retrieveBackendJobList` dans `JobeetJobTable` situé dans `lib/model/doctrine/JobeetJobTable.class.php`.

```
// lib/model/doctrine/JobeetJobTable.class.php
class JobeetJobTable extends Doctrine_Table
{
  public function retrieveBackendJobList(Doctrine_Query $q)
  {
    $rootAlias = $q->getRootAlias();
    $q->leftJoin($rootAlias . '.JobeetCategory c');
    return $q;
  }

  // ...
```

La méthode `retrieveBackendJobList()` ajoute une jointure entre les tables `job` et `category` et crée automatiquement l'objet catégorie lié à chaque emploi.

Le nombre de requêtes est maintenant réduit à quatre:



Configuration des vues du formulaire

La configuration des vues du formulaire se fait dans trois sections : `form`, `edit` et `new`. Elles ont tous les mêmes capacités de configuration et la section `form` existe seulement en tant que solution de repli pour les sections `edit` et `new`.

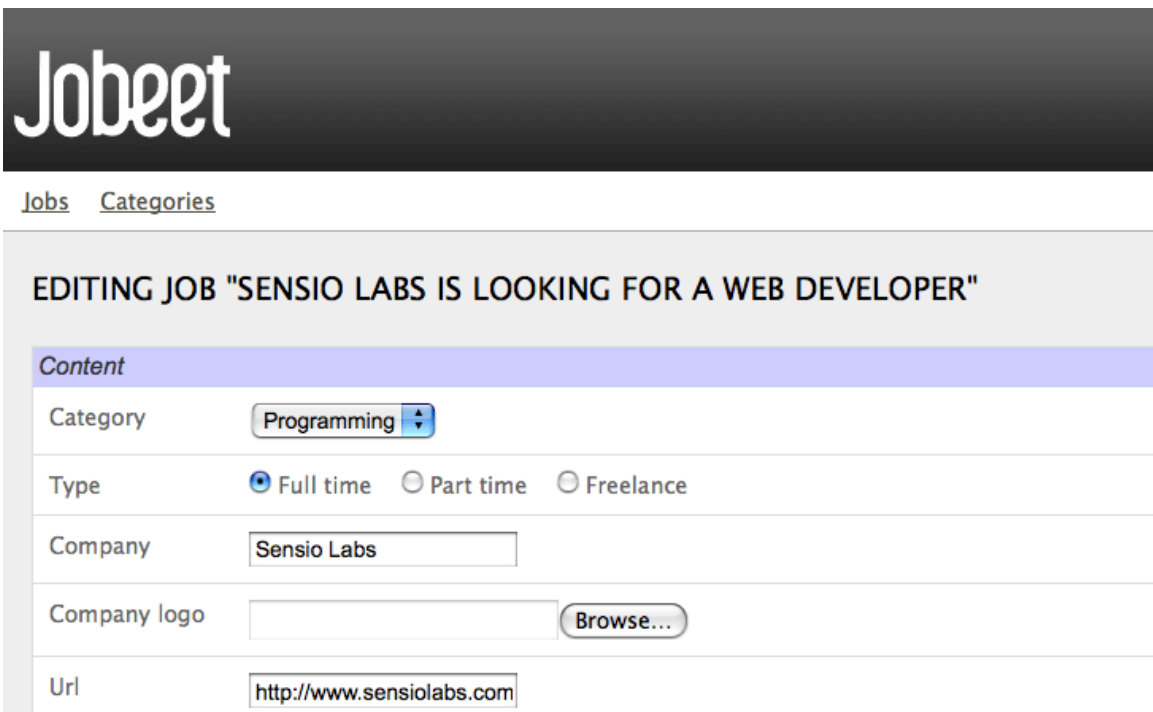
display

Comme pour la liste, vous pouvez changer l'ordre des champs affichés avec l'option `display`. Mais comme le formulaire affiché est définie par une classe, n'essayez pas de supprimer un champ car cela pourrait conduire à des erreurs inattendues.

L'option `display` pour les vues du formulaire peut également être utilisé pour organiser des champs en groupes :

```
# apps/backend/modules/job/config/generator.yml
config:
  form:
    display:
      Content: [category_id, type, company, logo, url, position, location,
description, how_to_apply, is_public, email]
      Admin:   [_generated_token, is_activated, expires_at]
```

La configuration ci-dessus définit deux groupes (`Content` et `Admin`), contenant chacune un sous-ensemble de champs du formulaire.



Les colonnes dans le groupe `Admin` n'apparaissent pas dans le navigateur pour le moment car elles ont été mis hors d'usage dans la définition du formulaire emploi. Elles apparaîtront dans quelques sections où nous aurons défini une classe de formulaire d'emplois personnalisée pour l'application admin.

L'admin generator a un support intégré pour la relation plusieurs vers plusieurs. Sur le formulaire de catégorie, vous disposez d'une entrée pour le nom, d'une pour le slug et une liste déroulante pour les affiliés connexes. Comme cela n'a pas de sens de modifier cette relation sur cette page, nous allons la supprimer :

```
// lib/form/doctrine/JobeetCategoryForm.class.php
class JobeetCategoryForm extends BaseJobeetCategoryForm
{
  public function configure()
  {
    unset($this['created_at'], $this['updated_at'], $this['jobeet_affiliates_list']);
  }
}
```

Les colonnes "Virtuelles"

Dans l'option `display` pour le formulaire d'emplois, le `_generated_token` commence par un caractère de soulignement (`_`). Cela signifie que le rendu de ce champ sera traitée par un partial

personnalisé nommé `_generated_token.php`.

Créer ce partial avec le contenu suivant :

```
// apps/backend/modules/job/templates/_generated_token.php
<div class="sf_admin_form_row">
  <label>Token</label>
  <?php echo $form->getObject()->getToken() ?>
</div>
```

Dans le partial, vous avez accès au formulaire actuel (`$form`) et l'objet connexe est accessible via la méthode `getObject()`.

Vous pouvez aussi déléguer le rendu d'un composant en faisant précéder le nom du champ par un tilde (~).

---

class

Comme le formulaire sera utilisé par les administrateurs, nous avons affiché plus d'informations que pour le formulaire emploi de l'utilisateur. Mais pour l'instant, certains d'entre eux ne figurent pas sur le formulaire car ils l'ont été enlevés dans la classe `JobeetJobForm`.

Pour avoir des formulaires différents pour le frontend et le backend, nous avons besoin de créer deux classes de formulaire. Nous allons créer une classe `BackendJobeetJobForm` qui étend la classe `JobeetJobForm`. Comme nous n'aurons pas les mêmes champs cachés, nous devons aussi un peu refactoriser la classe `JobeetJobForm` pour déplacer l'instruction `unset()` dans une méthode qui substituera dans `BackendJobeetJobForm` :

```
// lib/form/doctrine/JobeetJobForm.class.php
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    $this->removeFields();

    $this->validatorSchema['email'] = new sfValidatorAnd(array(
      $this->validatorSchema['email'],
      new sfValidatorEmail(),
    ));

    // ...
  }

  protected function removeFields()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['expires_at'], $this['is_activated'],
      $this['token']
    );
  }
}

// lib/form/doctrine/BackendJobeetJobForm.class.php
class BackendJobeetJobForm extends JobeetJobForm
{
  protected function removeFields()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['token']
    );
  }
}
```

La classe du formulaire par défaut utilisé par l'admin generator peut être surchargée en définissant l'option `class` :

```
# apps/backend/modules/job/config/generator.yml
config:
  form:
    class: BackendJobeetJobForm
```

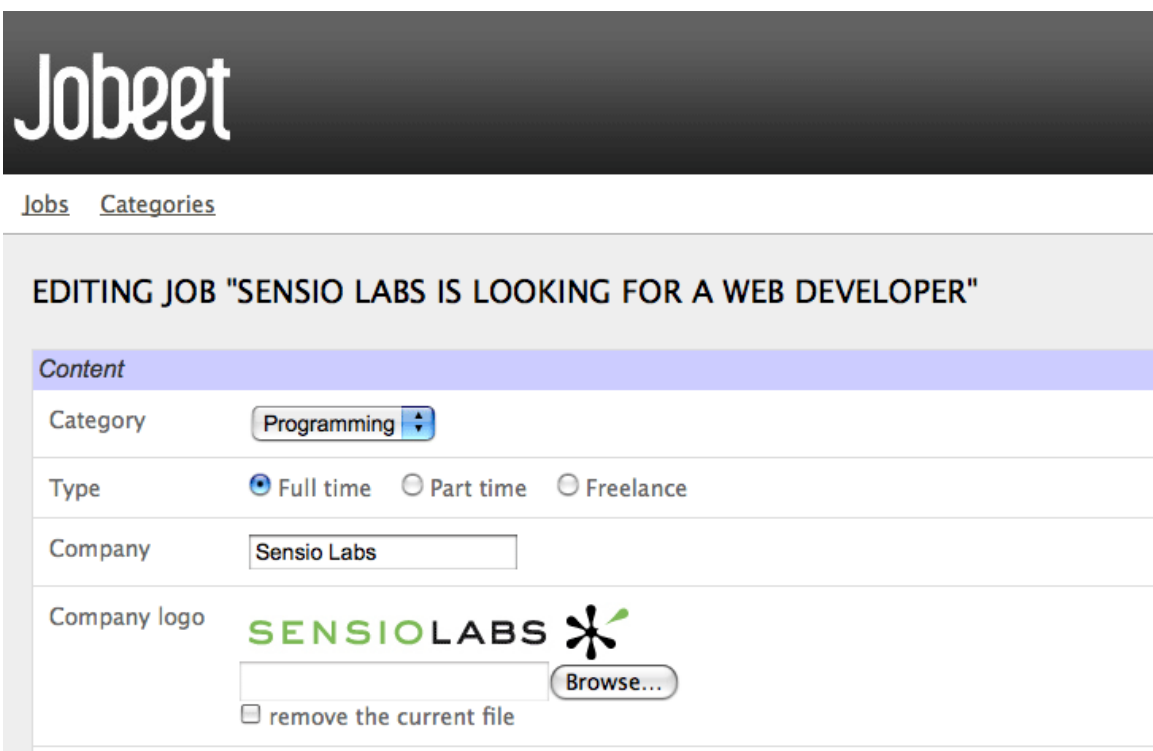
Comme nous avons ajouté une nouvelle classe, n'oubliez pas de vider le cache.

Le formulaire edit a toujours un petit ennui. Le logo de téléchargement ne s'affiche pas partout et vous ne pouvez pas supprimer l'actuel. Le widget `sfWidgetFormInputFileEditable` ajoute des possibilités d'édition à une simple saisie de fichier :

```
// lib/form/doctrine/BackendJobeetJobForm.class.php
class BackendJobeetJobForm extends JobeetJobForm
{
  public function configure()
  {
    parent::configure();

    $this->widgetSchema['logo'] = new sfWidgetFormInputFileEditable(array(
      'label'     => 'Company logo',
      'file_src'  => '/uploads/jobs/'.$this->getObject()->getLogo(),
      'is_image'  => true,
      'edit_mode' => !$this->isNew(),
      'template'  => '<div>%file%<br />%input%<br />%delete% %delete_label%</div>',
    ));

    $this->validatorSchema['logo_delete'] = new sfValidatorPass();
  }

  // ...
}
```

Le widget `sfWidgetFormInputFileEditable` a plusieurs options pour peaufiner ses caractéristiques et le rendu :

- `file_src`: Le chemin web pour télécharger le fichier
- `is_image`: Si `true`, le fichier sera rendu comme une image
- `edit_mode`: Si le formulaire est en mode édition ou non
- `with_delete`: S'il faut afficher la case à cocher pour supprimer
- `template`: Le Template à utiliser pour rendre le widget



Le look de l'admin generator peut être modifié très facilement car les Templates générés définissent beaucoup d'attributs `class` et `id`. Par exemple, le champ du logo peut être personnalisé en utilisant la classe `sf_admin_form_field_logo`. Chaque champ a également

une classe en fonction du type du champ, comme `sf_admin_text` ou `sf_admin_boolean`.

L'option `edit_mode` utilise la méthode `sfDoctrineRecord::isNew()`.

Elle retourne `true` si l'objet du modèle du formulaire est nouveau, sinon `false`. Ceci est d'une grande aide lorsque vous avez besoin d'avoir différents widgets ou validateurs en fonction du statut de l'objet incorporé.

Configuration des filtres

La configuration des filtres est tout à fait la même que la configuration des vues du formulaire. En fait, les filtres ne sont que des formulaires. Et comme pour les formulaires, les classes ont été générés par la tâche `doctrine:build --all`. Vous pouvez également les re-générer avec la tâche `doctrine:build --filters`.

Les classes de filtre de formulaire sont situées sous le répertoire `lib/filter/` et chaque classe du modèle est associée à une classe de filtre de formulaire (`JobeetJobFormFilter` pour `JobeetJobForm`).

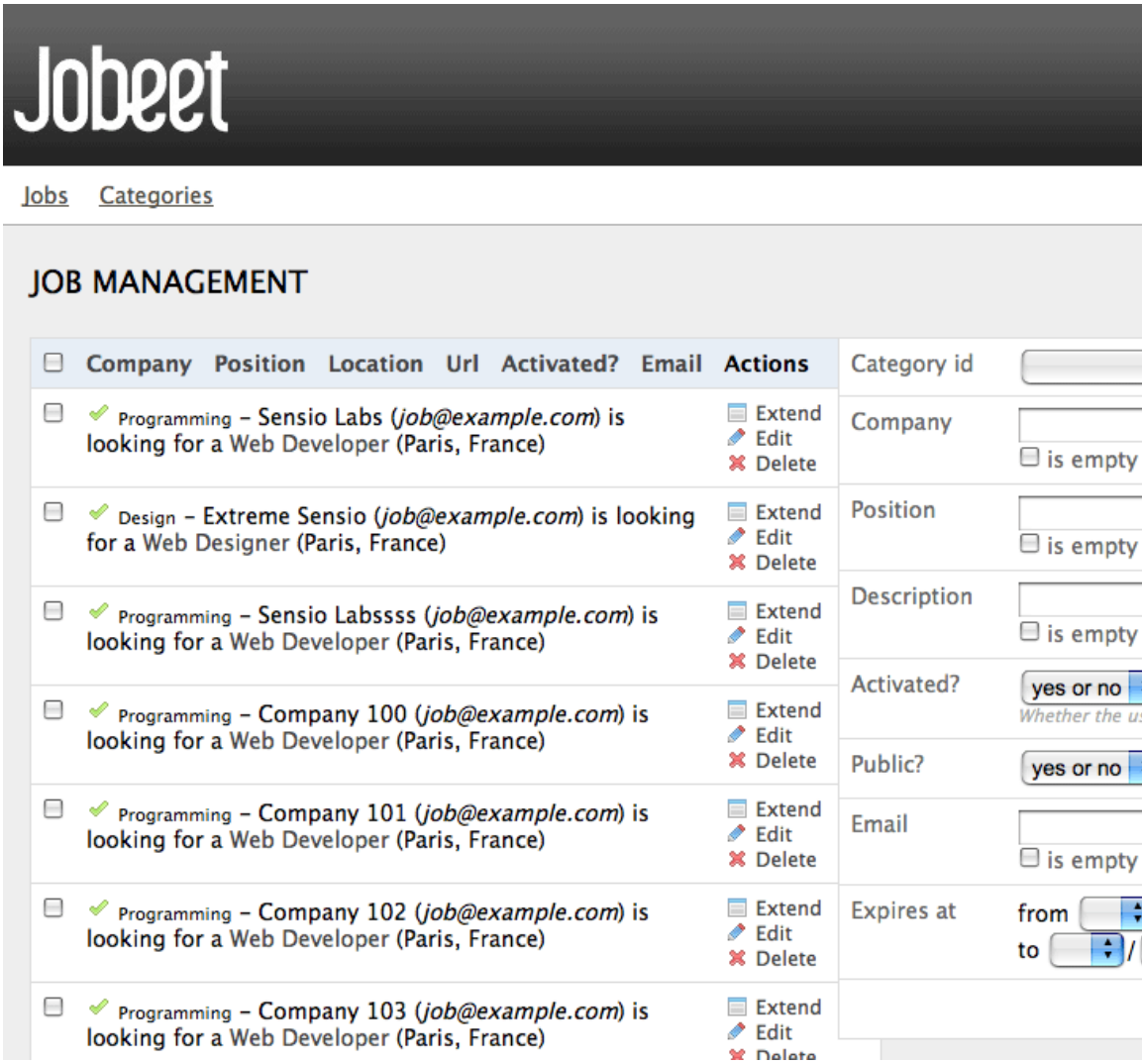
Supprimons-les complètement pour le module `category` :

```
# apps/backend/modules/category/config/generator.yml
config:
  filter:
    class: false
```

Pour le module `job`, nous allons supprimer certains d'entre eux :

```
# apps/backend/modules/job/config/generator.yml
filter:
  display: [category_id, company, position, description, is_activated, is_public,
email, expires_at]
```

Comme les filtres sont toujours facultatifs, il n'y a pas besoin de surcharger la classe du formulaire de filtre pour configurer les champs à afficher.



Personnalisation des actions

Lorsque la configuration n'est pas suffisante, vous pouvez ajouter de nouvelles méthodes pour la classe de l'action comme nous l'avons vu avec la fonctionnalité d'extension, mais vous pouvez aussi surcharger les méthodes de l'action générées :

Méthode	Description
`executeIndex()`	Action de la vue `list`
`executeFilter()`	Mettre à jour les filtres
`executeNew()`	Action de la vue `new`
`executeCreate()`	Créer un nouvel emploi
`executeEdit()`	Action de la vue `edit`
`executeUpdate()`	Mettre à jour un emploi
`executeDelete()`	Supprimer un emploi
`executeBatch()`	Executer une action batch
`executeBatchDelete()`	Executer l'action batch `_delete`
`processForm()`	Processer le formulaire emploi
`getFilters()`	Retourner le filtre actuel
`setFilters()`	Définir le filtre
`getPager()`	Retourner la pagination de la liste
`getPage()`	Obtenir la page de la pagination
`setPage()`	Définir la page de la pagination

buildCriteria()	Construire le Criteria pour la liste
addSortCriteria()	Ajouter le tri Criteria pour la liste
getSort()	Retourner la colonne triée actuelle
setSort()	Définit la colonne triée actuelle

Comme chaque méthode générée ne fait qu'une chose, il est facile de changer un comportement sans avoir à copier et coller trop de code.

Personnalisation des Templates

Nous avons vu comment personnaliser les templates générés grâce aux attributs `class` et `id` ajoutés par l'admin generator dans le code HTML.

Quant aux classes, vous pouvez également remplacer les Templates originaux. Comme les Templates sont des simples fichiers PHP et non des classes PHP, un Template peut être substituée en créant un Template du même nom dans le module (par exemple dans le répertoire `apps/backend/modules/job/templates/` pour le module de l'admin `job`) :

Template	Description
_assets.php	Rendre lees CSS et les JS pour les utiliser dans les Templates
_filters.php	Rendre la zone des filtres
_filters_field.php	Rendre un seul champ du filtre
_flashes.php	Rendre les messages flash
_form.php	Afficher le formulaire
_form_actions.php	Afficher les actions du formulaire
_form_field.php	Afficher un seul champ du formulaire
_form_fieldset.php	Afficher un jeu de champs du formulaire
_form_footer.php	Afficher le formulaire pied de page
_form_header.php	Afficher le formulaire d'entête
_list.php	Afficher la liste
_list_actions.php	Afficher les actions de la liste
_list_batch_actions.php	Afficher les actions batch de la liste
_list_field_boolean.php	Afficher un seul champ booléen dans la liste
_list_footer.php	Afficher le pied de page de la liste
_list_header.php	Afficher l'entête de la liste
_list_td_actions.php	Afficher les actions d'un objet pour une ligne
_list_td_batch_actions.php	Afficher le checkbox pour une ligne
_list_td_stacked.php	Afficher le layout stacked pour une ligne
_list_td_tabular.php	Afficher un seul champ pour la liste
_list_th_stacked.php	Afficher un seul nom de colonne pour l'entête
_list_th_tabular.php	Afficher un seul nom de colonne pour l'entête
_pagination.php	Afficher la pagination de la liste
editSuccess.php	Afficher la vue edit
indexSuccess.php	Afficher la vue list
newSuccess.php	Afficher la vue new

Configuration finale

La configuration finale pour l'admin de Jobeet admin se présente comme suit :

```
# apps/backend/modules/job/config/generator.yml
generator:
  class: sfDoctrineGenerator
```

```
  param:
    model_class:           JobeetJob
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          jobeet_job
    with_doctrine_route:   true

    config:
      actions: ~
      fields:
        is_activated: { label: Activated?, help: Whether the user has activated the
job, or not }
        is_public:    { label: Public? }
      list:
        title:         Job Management
        layout:        stacked
        display:       [company, position, location, url, is_activated, email]
        params:  |
          %%is_activated%% <small>%%JobeetCategory%%</small> - %%company%%
           (<em>%%email%%</em>) is looking for a %%=position%% (%%location%%)
        max_per_page:  10
        sort:          [expires_at, desc]
        batch_actions:
          _delete:    ~
          extend:     ~
        object_actions:
          extend:     ~
          _edit:      ~
          _delete:    ~
        actions:
          deleteNeverActivated: { label: Delete never activated jobs }
        table_method: retrieveBackendJobList
      filter:
        display: [category_id, company, position, description, is_activated,
is_public, email, expires_at]
      form:
        class:     BackendJobeetJobForm
        display:
          Content: [category_id, type, company, logo, url, position, location,
description, how_to_apply, is_public, email]
          Admin:   [_generated_token, is_activated, expires_at]
      edit:
        title: Editing Job "%%company%% is looking for a %%position%%"
      new:
        title: Job Creation

# apps/backend/modules/category/config/generator.yml
generator:
  class: sfDoctrineGenerator
  param:
    model_class:           JobeetCategory
    theme:                 admin
    non_verbose_templates: true
    with_show:             false
    singular:              ~
    plural:                ~
    route_prefix:          jobeet_category
    with_doctrine_route:   true

    config:
      actions: ~
      fields:  ~
      list:
        title:   Category Management
        display: [=name, slug]
        batch_actions: {}
        object_actions: {}
      filter:
        class: false
```

```
      form:
        actions:
          _delete: ~
          _list:   ~
          _save:   ~
      edit:
        title: Editing Category "%%name%%"
      new:
        title: New Category
```

Avec seulement ces deux fichiers de configuration, nous avons développé une interface backend idéal pour Jobeet en quelques minutes.

Vous savez déjà que lorsque quelque chose est configurable dans un fichier YAML, il y a aussi la possibilité d'utiliser du code PHP. Pour l'admin generator, vous pouvez modifier le fichier `apps/backend/modules/job/lib/jobGeneratorConfiguration.class.php`. Il vous donne les mêmes options que le fichier YAML mais avec une interface PHP. Pour apprendre les noms des méthodes, jetez un oeil à la classe de base générée en `cache/backend/dev/modules/autoJob/lib/BaseJobGeneratorConfiguration.class.php`.

Conclusion

En une heure seulement, nous avons construit entièrement une interface backend pour le projet Jobeet. Et dans l'ensemble, nous avons écrit au plus 50 lignes de code PHP. Pas trop mal pour de nombreuses fonctionnalités !

Demain, nous allons voir comment sécuriser l'application backend avec un identifiant et un mot de passe. Ce sera également l'occasion de parler de la classe user de symfony.

« Jour 11 : Testez votre formulaire

Jour 13 : L'utilisateur »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 13 : L'utilisateur

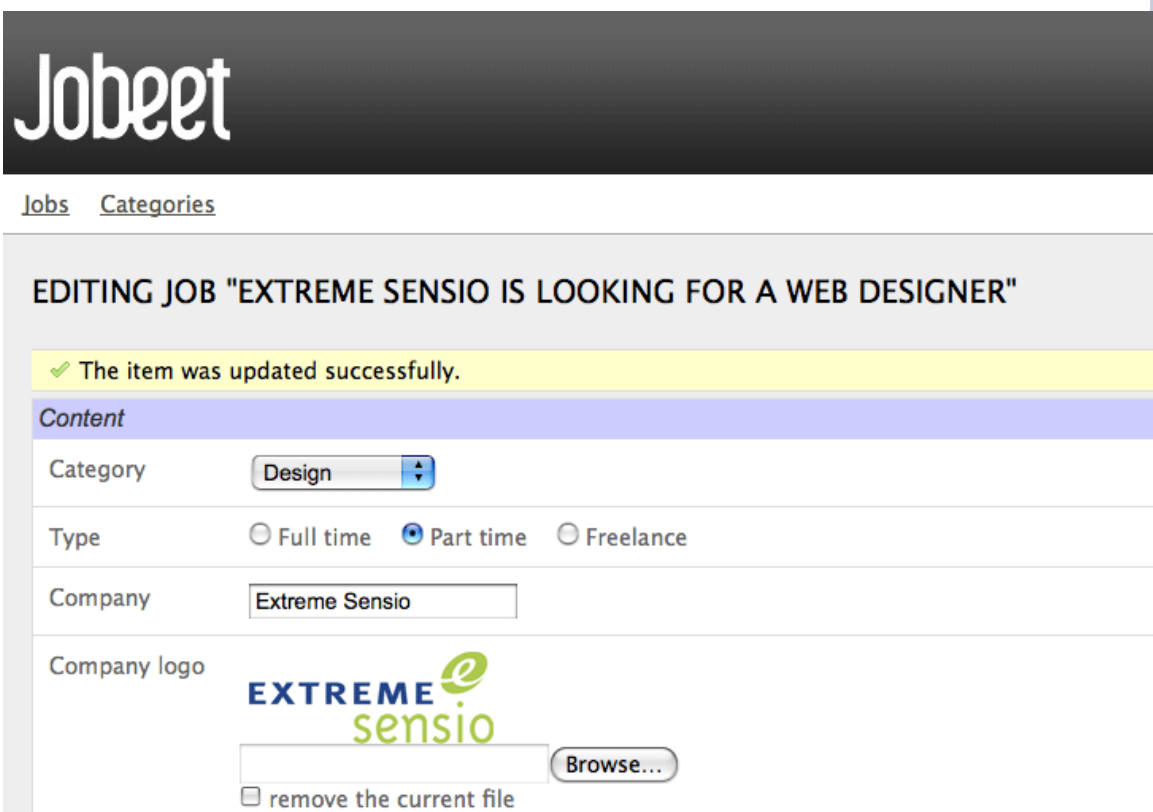
Hier a été emballé avec beaucoup d'informations. Avec très peu de lignes de code PHP, l'admin generator de symfony permet au développeur de créer des interfaces backend en quelques minutes.

Aujourd'hui, nous allons découvrir comment symfony gère les données persistantes entre les requêtes HTTP. Comme vous le savez, le protocole HTTP est sans état, ce qui signifie que chaque requête est indépendante de ses précédentes ou suivantes. Les sites web modernes ont besoin d'un moyen de maintenir les données entre les requêtes pour améliorer l'expérience utilisateur.

Une session d'utilisateur peut être identifiée à l'aide d'un cookie. Dans Symfony, le développeur n'a pas besoin de manipuler la session directement, mais utilise plutôt l'objet `sfUser`, qui représente l'utilisateur final de l'application.

Flashes d'utilisateur

Nous avons déjà vu l'objet user dans l'action avec des flashes. Un flash est un message éphémère stocké dans la session de l'utilisateur qui sera automatiquement supprimé après la requête suivante. Il est très utile lorsque vous avez besoin d'afficher un message à l'utilisateur après une redirection. L'admin generator utilise beaucoup les flashes pour afficher les informations à l'utilisateur chaque fois qu'un emploi est enregistré, effacé ou prolongé.



Un flash est défini en utilisant la méthode `setFlash()` de `sfUser`:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeExtend(sfWebRequest $request)
{
  $request->checkCSRFProtection();
```

```
  $job = $this->getRoute()->getObject();
  $this->forward404Unless($job->extend());

  $this->getUser()->setFlash('notice', sprintf('Your job validity has been extended
until %s.', $job->getDateTimeObject('expires_at')->format('m/d/Y')));

  $this->redirect($this->generateUrl('job_show_user', $job));
}
```

Le premier argument est l'identifiant du flash et le second est le message à afficher. Vous pouvez définir les flashes que vous voulez, mais `notice` et `error` sont les deux des algorithmes les plus utilisés (ils sont largement utilisés par l'admin generator).

Il appartient au développeur d'inclure le message flash dans les Templates. Pour Jobeet, ils sont affichés par le `layout.php` :

```
// apps/frontend/templates/layout.php
<?php if ($sf_user->hasFlash('notice')): ?>
  <div class="flash_notice"><?php echo $sf_user->getFlash('notice') ?></div>
<?php endif ?>

<?php if ($sf_user->hasFlash('error')): ?>
  <div class="flash_error"><?php echo $sf_user->getFlash('error') ?></div>
<?php endif ?>
```

Dans un Template, l'utilisateur est accessible via la variable spéciale `$sf_user`.

Certains objets de symfony sont toujours accessibles dans les Templates, sans qu'il soit nécessaire de les passer explicitement à l'action : `$sf_request`, `$sf_user` et `$sf_response`.

Les attributs d'utilisateur

Malheureusement, les histoires d'utilisateurs de Jobeet n'ont aucune exigence qui incluent le stockage de quelque chose dans la session utilisateur. Ajoutons donc une nouvelle exigence : pour faciliter la navigation dans l'emploi, les trois derniers emplois vus par l'utilisateur doivent être affiché dans le menu avec des liens pour revenir à la page des emplois par la suite.

Quand un utilisateur accède à une page d'emploi, l'objet affiché emploi doit être ajoutée dans l'historique de l'utilisateur et stocké dans la session :

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();

    // fetch jobs already stored in the job history
    $jobs = $this->getUser()->getAttribute('job_history', array());

    // add the current job at the beginning of the array
    array_unshift($jobs, $this->job->getId());

    // store the new job history back into the session
    $this->getUser()->setAttribute('job_history', $jobs);
  }

  // ...
}
```

Nous aurions pu en pratique stocker les objets JobeetJob directement dans la session. Cela est fortement déconseillé, car les variables de session sont sérialisées entre les requêtes. Et lorsque la session est chargée, les objets JobeetJob sont dé-sérialisés et peuvent être "bloqués", s'ils ont été modifiés ou supprimés entre-temps.

`getAttribute()`, `setAttribute()`

Pour un identifiant donné, la méthode `sfUser::getAttribute()` récupère les valeurs de la session de l'utilisateur. Inversement, la méthode `setAttribute()` stocke toutes les variables

PHP dans la session pour un identifiant donné.

La méthode `getAttribute()` prend également une valeur facultative par défaut qu'elle retourne si l'identifiant n'est pas encore défini.

La valeur par défaut prise par la méthode `getAttribute()` est un raccourci pour :

```
if (!$value = $this->getAttribute('job_history'))
{
  $value = array();
}
```

La classe *myUser*

Afin de mieux respecter la séparation des préoccupations, nous allons déplacer le code dans la classe `myUser`. La classe `myUser` substitue la classe de base de symfony par défaut `sfUser` avec les comportements spécifiques de l'application :

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();

    $this->getUser()->addJobToHistory($this->job);
  }

  // ...
}

// apps/frontend/lib/myUser.class.php
class myUser extends sfBasicSecurityUser
{
  public function addJobToHistory(JobeetJob $job)
  {
    $ids = $this->getAttribute('job_history', array());

    if (!in_array($job->getId(), $ids))
    {
      array_unshift($ids, $job->getId());

      $this->setAttribute('job_history', array_slice($ids, 0, 3));
    }
  }
}
```

Le code a également été modifiées afin de prendre en compte toutes les exigences :

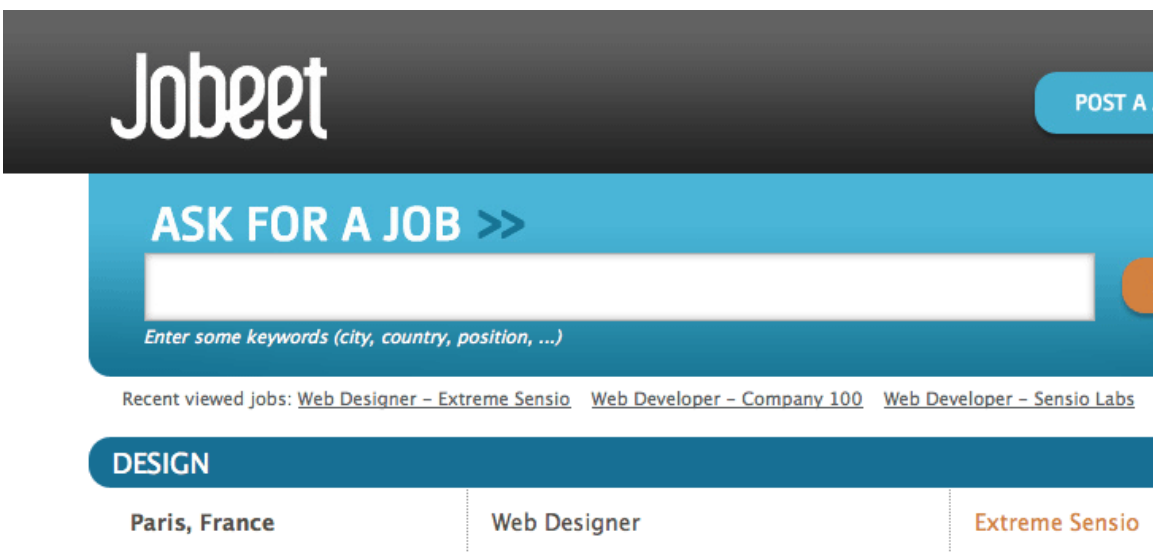
- `!in_array($job->getId(), $ids)` : Un emploi ne peut être stocké deux fois dans l'historique
- `array_slice($ids, 0, 3)` : Seuls les trois derniers emplois vus par l'utilisateur sont affichés

Dans le layout, ajoutez le code suivant avant que la variable `$sf_content` soit affichée :

```
// apps/frontend/templates/layout.php
<div id="job_history">
  Recent viewed jobs:
  <ul>
    <?php foreach ($sf_user->getJobHistory() as $job): ?>
      <li>
        <?php echo link_to($job->getPosition().' - '.$job->getCompany(),
'job_show_user', $job) ?>
      </li>
    <?php endforeach ?>
  </ul>
</div>

<div class="content">
  <?php echo $sf_content ?>
</div>
```

La mise en page utilise une nouvelle méthode `getJobHistory()` pour récupérer l'historique des emplois actuels :

```
// apps/frontend/lib/myUser.class.php
class myUser extends sfBasicSecurityUser
{
  public function getJobHistory()
  {
    $ids = $this->getAttribute('job_history', array());

    if (!empty($ids))
    {
      return Doctrine_Core::getTable('JobeetJob')
        ->createQuery('a')
        ->whereIn('a.id', $ids)
        ->execute()
      ;
    }

    return array();
  }

  // ...
}
```



sfParameterHolder

Pour compléter l'API de l'historique des emplois, nous allons ajouter une méthode pour réinitialiser l'historique :

```
// apps/frontend/lib/myUser.class.php
class myUser extends sfBasicSecurityUser
{
  public function resetJobHistory()
  {
    $this->getAttributeHolder()->remove('job_history');
  }

  // ...
}
```

Les attributs de l'utilisateur sont gérés par un objet de la classe `sfParameterHolder`. Les méthodes `getAttribute()` et `setAttribute()` sont des méthodes proxy pour `getParameterHolder()->get()` et `getParameterHolder()->set()`. Comme la méthode `remove()` n'a pas de méthode de proxy dans `sfUser`, vous devez utiliser l'objet titulaire du paramètre directement.

La classe `sfParameterHolder` est aussi utilisé par `sfRequest` pour stocker ses paramètres.

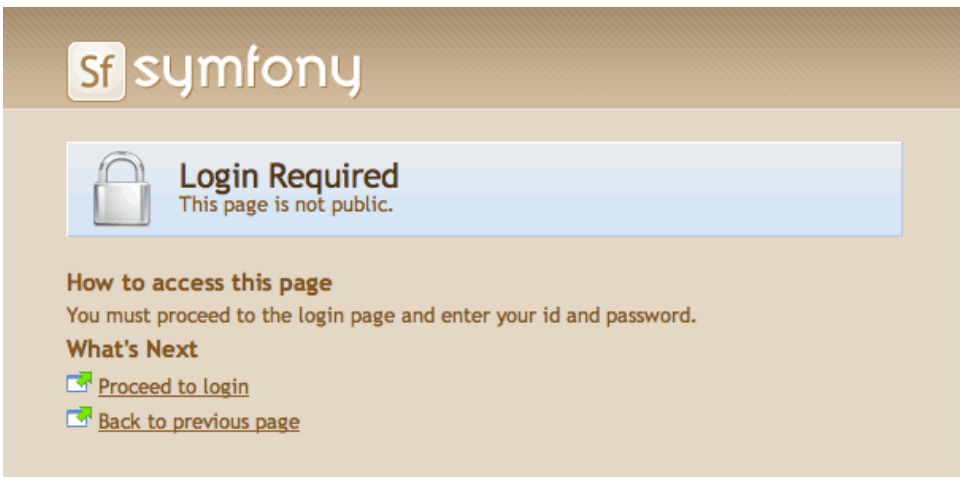
Sécurité de l'application

Authentification

Comme beaucoup d'autres fonctionnalités symfony, la sécurité est gérée par un fichier YAML, `security.yml`. Par exemple, vous pouvez trouver la configuration par défaut pour l'application backend dans le répertoire `config/` :

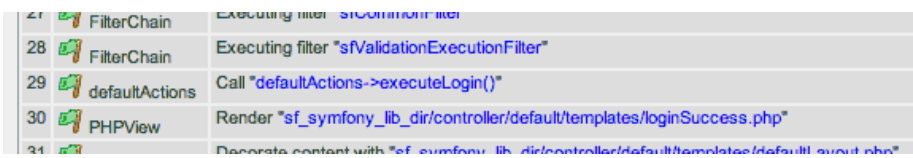
```
# apps/backend/config/security.yml
default:
  is_secure: false
```

Si vous passez l'entrée `is_secure` à `true`, l'ensemble de l'application backend nécessitera que l'utilisateur soit authentifié.



Dans un fichier YAML, un booléen peut être exprimée avec les chaines `true` et `false`.

Si vous jetez un oeil sur les journaux dans le web debug toolbar, vous verrez que la méthode `executeLogin()` de la classe `defaultActions` est appelée à chaque page que vous essayez d'accéder.



Quand un utilisateur non-authentifié tente d'accéder à une action sécurisé, symfony transmet la requête à l'action `login` configuré dans `settings.yml` :

```
all:
  .actions:
    login_module: default
    login_action: login
```

Il n'est pas possible de sécuriser l'action de connexion afin d'éviter une récursion infinie.

Comme nous l'avons vu pendant le jour 4, le même fichier de configuration peut être défini dans plusieurs endroits. C'est également le cas pour `security.yml`. Pour sécuriser ou dé-sécuriser seulement une seule action ou un module complet, créer un `security.yml` dans le répertoire `config/` du module :

```
index:
  is_secure: false

all:
  is_secure: true
```

Par défaut, la classe `myUser` étend `sfBasicSecurityUser`, et non `sfUser`. `sfBasicSecurityUser` fournit des méthodes supplémentaires pour gérer l'authentification des

utilisateurs et l'autorisation.

Pour gérer l'authentification utilisateur, utilisez les méthodes `isAuthenticated()` et `setAuthenticated()` :

```
if (!$this->getUser()->isAuthenticated())
{
  $this->getUser()->setAuthenticated(true);
}
```

Autorisation

Lorsqu'un utilisateur est authentifié, l'accès à certaines actions peuvent être encore plus limité en définissant les **credentials**. Un utilisateur doit disposer des droits nécessaires pour accéder à la page :

```
default:
  is_secure:   false
  credentials: admin
```

Le système de droits de symfony est assez simple et puissant. Un credential peut représenter tout ce que vous devez décrire au modèle de sécurité de l'application (comme des groupes ou des autorisations).

Les Credentials complexes

L'entrée `credentials` de `security.yml` supporte des opérations booléennes pour décrire les exigences des credentials complexes.

Si un utilisateur doit avoir un credential A **et** B, enveloppez-les avec une paire de crochets :

```
index:
  credentials: [A, B]
```

Si un utilisateur doit avoir un credential A **ou** B, enveloppez-les avec deux paires de crochets :

```
index:
  credentials: [[A, B]]
```

Vous pouvez même combiner entre parenthèses pour décrire tout type d'expression booléenne avec n'importe quel nombre de credentials.

Pour gérer les credentials d'utilisateur, `sfBasicSecurityUser` fournit plusieurs méthodes :

```
// Add one or more credentials
$user->addCredential('foo');
$user->addCredentials('foo', 'bar');

// Check if the user has a credential
echo $user->hasCredential('foo');                       =>   true

// Check if the user has both credentials
echo $user->hasCredential(array('foo', 'bar'));         =>   true

// Check if the user has one of the credentials
echo $user->hasCredential(array('foo', 'bar'), false); =>   true

// Remove a credential
$user->removeCredential('foo');
echo $user->hasCredential('foo');                       =>   false

// Remove all credentials (useful in the logout process)
$user->clearCredentials();
echo $user->hasCredential('bar');                       =>   false
```

Pour le backend de Jobeet, nous n'utiliserons pas les credentials car nous avons qu'un seul profil: l'administrateur.

Plugins

Comme nous n'aimons pas à réinventer la roue, nous ne développerons pas l'action de connexion à partir de zéro. Au lieu de cela, nous allons installer un **plugin symfony**.

Une des grandes forces du framework symfony est l'écosystème de plugin. Comme nous le verrons dans les prochains jours, il est très facile de créer un plugin. Il est aussi assez puissant, car un plugin peut contenir n'importe quoi, de la configuration aux modules et aux ressources.

Aujourd'hui, nous allons installer sfDoctrineGuardPlugin pour sécuriser l'application backend :

```
$ php symfony plugin:install sfDoctrineGuardPlugin
```

La tâche `plugin:install` installe un plugin par son nom. Tous les plugins sont stockés dans le répertoire `plugins/` et chacun a son propre répertoire nommé d'après le nom du plugin.

PEAR doit être installé pour que la tâche `plugin:install` fonctionne.

Lorsque vous installez un plugin avec la tâche `plugin:install`, symfony installe sa dernière version stable. Pour installer une version spécifique d'un plugin, passer l'option `--release`.

La page du plugin liste toutes les versions disponibles regroupés selon la version de symfony.

Comme un plugin est autonome dans un répertoire, vous pouvez également télécharger le package à partir du site web de symfony et le décompressez, ou bien faire un lien `svn:externals`, grâce à son dépôt Subversion.

La tâche `plugin:install` active automatiquement le(s) plugin(s) et elle les installe en mettant à jour automatiquement le fichier `ProjectConfiguration.class.php`. Mais si vous installez un plugin via Subversion ou en téléchargeant son archive, vous devez l'activer à la main dans `ProjectConfiguration.class.php` :

```
// config/ProjectConfiguration.class.php
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    $this->enablePlugins(array(
      'sfDoctrinePlugin',
      'sfDoctrineGuardPlugin'
    ));
  }
}
```

Sécurité du backend

Chaque plugin a un fichier README qui explique comment le configurer.

Voyons comment configurer le nouveau plugin. Comme le plugin fournit plusieurs nouvelles classes du modèle pour gérer les utilisateurs, les groupes et les autorisations, vous devez reconstruire votre modèle :

```
$ php symfony doctrine:build --all --and-load --no-confirmation
```

N'oubliez pas que la tâche `doctrine:build --all --and-load` supprime toutes les tables existantes avant de les re-créer. Pour éviter cela, vous pouvez construire les modèles, les formulaires, et les filtres, puis, créer les nouvelles tables en exécutant les instructions SQL générées stockées dans `data/sql/`.

Comme `sfDoctrineGuardPlugin` ajoute plusieurs méthodes à la classe user, vous avez besoin de changer la classe de base de `myUser` en `sfGuardSecurityUser` :

```
// apps/backend/lib/myUser.class.php
class myUser extends sfGuardSecurityUser
{
}
```

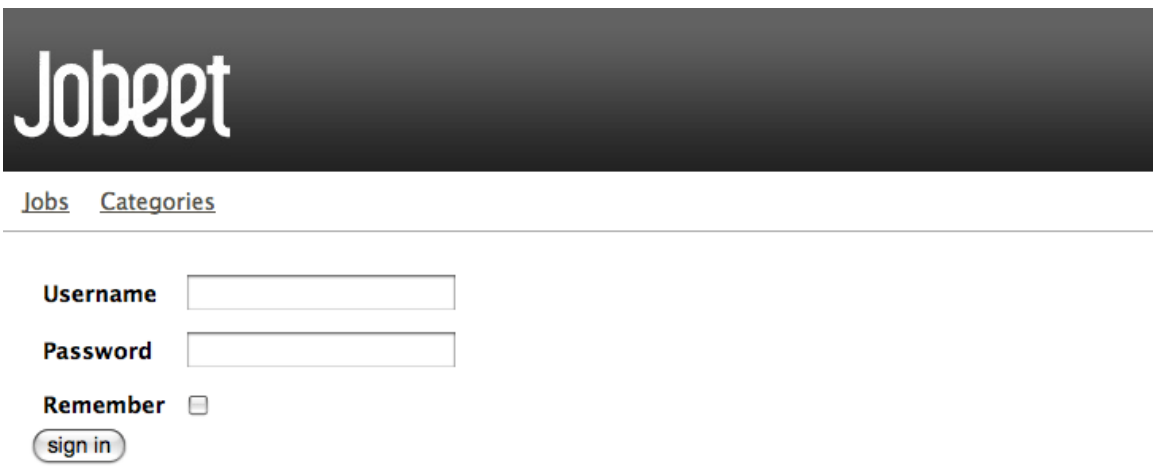
`sfDoctrineGuardPlugin` fournit une action `signin`dans le module `sfGuardAuth` pour authentifier les utilisateurs.

Editez le fichier `settings.yml` pour changer l'action par défaut utilisée pour la page du login :

```
# apps/backend/config/settings.yml
all:
  .settings:
    enabled_modules: [default, sfGuardAuth]
```

```
    # ...

  .actions:
    login_module:    sfGuardAuth
    login_action:    signin

    # ...
```

Comme les plugins sont partagées par toutes les applications d'un projet, vous devez explicitement activer les modules que vous souhaitez utiliser en les ajoutant dans le paramètre `enabled_modules|enabled_modules` (Paramètre).



La dernière étape est de créer un utilisateur administrateur :

```
$ php symfony guard:create-user fabien SecretPass
$ php symfony guard:promote fabien
```

Le `sfGuardPlugin` fournit des tâches pour gérer les utilisateurs, les groupes et les autorisations par la ligne de commande. Utilisez la tâche `list` pour lister toutes les tâches qui appartiennent à l'espace de nom de guard :

```
$ php symfony list guard
```

Lorsque l'utilisateur n'est pas authentifié, nous avons besoin de masquer la barre de menu :

```
// apps/backend/templates/layout.php
<?php if ($sf_user->isAuthenticated()): ?>
  <div id="menu">
    <ul>
      <li><?php echo link_to('Jobs', 'jobeet_job') ?></li>
      <li><?php echo link_to('Categories', 'jobeet_category') ?></li>
    </ul>
  </div>
<?php endif ?>
```

Et lorsque l'utilisateur est authentifié, nous avons besoin d'ajouter un lien de déconnexion dans le menu :

```
// apps/backend/templates/layout.php
<li><?php echo link_to('Logout', 'sf_guard_signout') ?></li>
```

Pour lister toutes les routes fournies par `sfGuardPlugin`, utiliser la tâche `app:routes`.

Pour améliorer, encore plus, le backend de Jobeet, nous allons ajouter un nouveau module pour gérer les utilisateurs administrateurs. Heureusement, `sfGuardPlugin` fournit un tel module. Comme pour le module `sfGuardAuth`, vous devez l'activer dans `settings.yml` :

```
// apps/backend/config/settings.yml
all:
  .settings:
```

```
    enabled_modules: [default, sfGuardAuth, sfGuardUser]
```

Ajoutez un lien dans le menu :

```
// apps/backend/templates/layout.php
<li><?php echo link_to('Users', 'sf_guard_user') ?></li>
```

Nous avons finis !

Tester l'utilisateur

Le tutoriel d'aujourd'hui n'est pas terminée tant que nous n'avons pas encore parlé des tests utilisateurs. Comme le navigateur symfony simule les cookies, il est assez facile de tester les comportements des utilisateurs en utilisant le testeur intégré `sfTesterUser`.

Actualisons les tests fonctionnels pour le menu des fonctions que nous avons ajouté aujourd'hui. Ajoutez le code suivant à la fin de la tâche des tests fonctionnels du module `job` :

```
// test/functional/frontend/jobActionsTest.php
$browser->
  info('4 - User job history')->

  loadData()->
  restart()->

  info('  4.1 - When the user access a job, it is added to its history')->
  get('/')->
  click('Web Developer', array(), array('position' => 1))->
  get('/')->
  with('user')->begin()->
    isAttribute('job_history', array($browser->getMostRecentProgrammingJob()-
>getId()))->
  end()->

  info('  4.2 - A job is not added twice in the history')->
  click('Web Developer', array(), array('position' => 1))->
  get('/')->
  with('user')->begin()->
    isAttribute('job_history', array($browser->getMostRecentProgrammingJob()-
>getId()))->
  end()
;
```

Pour faciliter le test, nous avons d'abord recharger les données des jeux de test et redémarrer le navigateur pour commencer par une session propre.

La méthode `isAttribute()` vérifie un attribut utilisateur donné.

Le testeur `sfTesterUser` fournit également les méthodes `isAuthenticated()` et `hasCredential()` pour tester les authentifications et les autorizations de l'utilisateur.

À demain

Les classes d'utilisateurs de symfony sont un moyen agréable pour la gestion des sessions PHP. Couplé avec l'excellent système de plugin de symfony et le plugin `sfGuardPlugin`, nous avons été en mesure de sécuriser le backend de Jobeet en quelques minutes. Et nous avons même

ajouté une interface propre pour gérer nos utilisateurs administrateurs gratuitement, grâce aux modules fournis par le plugin.

« Jour 12 : L'Admin Generator

Jour 14 : Les Flux »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 14 : Les Flux

Hier, vous avez commencé à développer votre première application symfony. Ne vous arrêtez pas maintenant. Car vous en apprendrez davantage sur symfony, essayez d'ajouter de nouvelles fonctionnalités à votre application, l'héberger quelque part, et la partager avec la communauté.

Passons à quelque chose de complètement différent aujourd'hui.

Si vous êtes à la recherche d'un emploi, vous aurez probablement besoin d'être informé dès qu'un nouveau poste est affiché. Parce qu'il n'est pas très pratique de vérifier le site web toutes les heures, nous allons ajouter plusieurs flux d'emploi aujourd'hui pour que nos utilisateurs Jobeet soient mis au courant.

Les formats

Le framework symfony a un support natif pour les formats et les mime-types. Cela signifie que le même modèle et contrôleur peut avoir différents Templates basés sur le format requêté. Le format par défaut est HTML, mais symfony supporte plusieurs autres formats comme `txt`, `js`, `css`, `json`, `xml`, `rdf`, ou `atom`.

Le format peut être définie en utilisant la méthode `setRequestFormat()` de l'objet de la requête :

```
$request->setRequestFormat('xml');
```

Mais la plupart du temps, le format est incorporé dans l'URL. Dans ce cas, symfony va le mettre pour vous, si la variable spéciale `sf_format` est utilisé dans la route correspondante. Pour la liste des emplois, l'URL de la liste est :

```
http://www.jobeet.com.localhost/frontend_dev.php/job
```

Cette URL est équivalent à :

```
http://www.jobeet.com.localhost/frontend_dev.php/job.html
```

Les deux URL sont équivalentes car les routes générés par la classe `sfDoctrineRouteCollection` ont le `sf_format` comme extension et parce que le HTML est le format par défaut. Vous pouvez le vérifier vous-même en exécutant la tâche `app:routes` :

```
~/work/jobeet $ ./symfony app:routes frontend
>> app       Current routes for application "frontend"
Name          Method Pattern
category      ANY    /category/:slug
job           GET    /job.:sf_format
job_new       GET    /job/new.:sf_format
job_create    POST   /job.:sf_format
job_edit      GET    /job/:token/edit.:sf_format
job_update    PUT    /job/:token.:sf_format
job_delete    DELETE /job/:token.:sf_format
job_show      GET    /job/:token.:sf_format
job_publish   PUT    /job/:token/publish.:sf_format
job_extend    PUT    /job/:token/extend.:sf_format
job_show_user GET    /job/:company_slug/:location_slug/:id/:position_slug
homepage      ANY    /
~/work/jobeet $
```

Les flux

Le flux du dernier emploi

Le support de différents formats est aussi facile que la création de Templates différents. Pour créer un flux Atom pour les derniers emplois, créez un Template `indexSuccess.atom.php` :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.atom.php -->
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Jobeet</title>
  <subtitle>Latest Jobs</subtitle>
  <link href="" rel="self"/>
  <link href=""/>
  <updated></updated>
  <author><name>Jobeet</name></author>
  <id>Unique Id</id>

  <entry>
    <title>Job title</title>
    <link href="" />
    <id>Unique id</id>
    <updated></updated>
    <summary>Job description</summary>
    <author><name>Company</name></author>
  </entry>
</feed>
```

Les noms des Templates

Comme `html` est le format le plus couramment utilisé pour les applications web, il peut être omis du nom du Template. Les deux Templates `indexSuccess.php` et `indexSuccess.html.php` sont équivalents et symfony utilise le premier qu'il trouve.

Pourquoi les Templates par défaut sont suffixés avec Success ? Une action ne peut retourner une valeur pour indiquer le Template à rendre. Si l'action renvoie rien, il est équivalent au code suivant :

```
return sfView::SUCCESS; // == 'Success'
```

Si vous voulez changer le suffixe, retournez simplement autre chose :

```
return sfView::ERROR; // == 'Error'

return 'Foo';
```

Comme nous l'avons vu dans la journée précédente, le nom du Template peut également être modifié en utilisant la méthode `setTemplate()` :

```
$this->setTemplate('foo');
```

Par défaut, symfony va changer le `Content-Type` de la réponse en fonction du format, et pour tous les formats non-HTML, le layout est désactivé. Pour un flux Atom, symfony change le `Content-Type` en `application/atom+xml; charset=utf-8`.

Dans le pied de page de Jobeet, mettez à jour le lien vers le flux :

```
<!-- apps/frontend/templates/layout.php -->
<li class="feed">
  <a href="<?php echo url_for('job', array('sf_format' => 'atom')) ?>">Full feed</a>
</li>
```

L'URI interne est la même que pour la liste `job` avec le `sf_format` ajouté comme une variable.

Ajoutez un balise `<link>` dans la section d'entête du layout pour permettre de découvrir par le navigateur automatiquement notre flux :

```
<!-- apps/frontend/templates/layout.php -->
<link rel="alternate" type="application/atom+xml" title="Latest Jobs"
  href="<?php echo url_for('job', array('sf_format' => 'atom'), true) ?>" />
```

Pour l'attribut `href` du lien, une URL (Absolue) est utilisée grâce au second argument du helper `url_for()`.

Remplacer l'entête du Template Atom avec le code suivant :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.atom.php -->
<title>Jobeet</title>
<subtitle>Latest Jobs</subtitle>
```

```
<link href="<?php echo url_for('job', array('sf_format' => 'atom'), true) ?>"
rel="self"/>
<link href="<?php echo url_for('homepage', true) ?>"/>
<updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
Doctrine_Core::getTable('JobeetJob')->getLatestPost()-
>getDateTimeObject('created_at')->format('U')) ?></updated>
<author>
  <name>Jobeet</name>
</author>
<id><?php echo sha1(url_for('job', array('sf_format' => 'atom'), true)) ?></id>
```

Notez l'utilisation du U comme un argument pour `format()` pour obtenir la date comme un timestamp. Pour obtenir la date du dernier post, créez la méthode `getLatestPost()` :

```
// lib/model/doctrine/JobeetJobTable.class.php
class JobeetJobTable extends Doctrine_Table
{
  public function getLatestPost()
  {
    $q = Doctrine_Query::create()
      ->from('JobeetJob j');
    $this->addActiveJobsQuery($q);

    return $q->fetchOne();
  }

  // ...
}
```

Les entrées du flux peuvent être générées par le code suivant :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.atom.php -->
<?php use_helper('Text') ?>
<?php foreach ($categories as $category): ?>
  <?php foreach ($category->getActiveJobs(sfConfig::get('app_max_jobs_on_homepage'))
as $job): ?>
    <entry>
      <title>
        <?php echo $job->getPosition() ?> (<?php echo $job->getLocation() ?>)
      </title>
      <link href="<?php echo url_for('job_show_user', $job, true) ?>" />
      <id><?php echo sha1($job->getId()) ?></id>
      <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ', $job-
>getDateTimeObject('created_at')->format('U')) ?></updated>
      <summary type="xhtml">
       <div xmlns="http://www.w3.org/1999/xhtml">
         <?php if ($job->getLogo()): ?>
           <div>
             <a href="<?php echo $job->getUrl() ?>">
               <img src="http://<?php echo $sf_request-
>getHost().'/uploads/jobs/'.$job->getLogo() ?>"
                    alt="<?php echo $job->getCompany() ?> logo" />
             </a>
           </div>
         <?php endif ?>

         <div>
           <?php echo simple_format_text($job->getDescription()) ?>
         </div>

         <h4>How to apply?</h4>

         <p><?php echo $job->getHowToApply() ?></p>
       </div>
      </summary>
      <author>
        <name><?php echo $job->getCompany() ?></name>
      </author>
    </entry>
  <?php endforeach ?>
<?php endforeach ?>
```

La méthode `getHost()` de l'objet de la requête (`$sf_request`) retourne l'hôte actuel, qui est

très pratique pour créer un lien absolu pour le logo de l'entreprise.

Lors de la création d'un flux, le débogage est plus facile si vous utilisez des outils en ligne de commande comme `curl` ou `wget`, car vous voyez le contenu actuel du flux.

Les derniers emplois dans le flux de la catégorie

L'un des objectifs de Jobeet est d'aider les gens à trouver un emploi plus ciblées. Donc, nous devons fournir un flux pour chaque catégorie.

D'abord, nous allons mettre à jour la route `category` pour ajouter le support de différents formats :

```
// apps/frontend/config/routing.yml
category:
  url:     /category/:slug.:sf_format
  class:   sfDoctrineRoute
  param:   { module: category, action: show, sf_format: html }
  options: { model: JobeetCategory, type: object }
  requirements:
    sf_format: (?:html|atom)
```

Maintenant, la route `category` comprend à la fois les formats `html` et `atom`. Mettez à jour les flux de la catégorie dans les Templates :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<div class="feed">
  <a href="<?php echo url_for('category', array('sf_subject' => $category, 'sf_format'
=> 'atom')) ?>">Feed</a>
</div>

<!-- apps/frontend/modules/category/templates/showSuccess.php -->
<div class="feed">
  <a href="<?php echo url_for('category', array('sf_subject' => $category, 'sf_format'
=> 'atom')) ?>">Feed</a>
</div>
```

La dernière étape consiste à créer le Template `showSuccess.atom.php`. Mais comme ce flux sera également la liste des emplois, nous pouvons refactoriser le code qui génère les entrées du flux en créant un partial `_list.atom.php`. Comme pour le format `html`, les partials ont un format spécifique :

```
<!-- apps/frontend/modules/job/templates/_list.atom.php -->
<?php use_helper('Text') ?>
```

```
<?php foreach ($jobs as $job): ?>
  <entry>
    <title><?php echo $job->getPosition() ?> (<?php echo $job->getLocation() ?
>)</title>
    <link href="<?php echo url_for('job_show_user', $job, true) ?>" />
    <id><?php echo sha1($job->getId()) ?></id>
      <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ', $job-
>getDateTimeObject('created_at')->format('U')) ?></updated>
    <summary type="xhtml">
     <div xmlns="http://www.w3.org/1999/xhtml">
       <?php if ($job->getLogo()): ?>
         <div>
           <a href="<?php echo $job->getUrl() ?>">
             <img src="http://<?php echo $sf_request->getHost().'/uploads/jobs/'.$job-
>getLogo() ?>"
                  alt="<?php echo $job->getCompany() ?> logo" />
           </a>
         </div>
       <?php endif ?>

       <div>
         <?php echo simple_format_text($job->getDescription()) ?>
       </div>

       <h4>How to apply?</h4>

       <p><?php echo $job->getHowToApply() ?></p>
     </div>
    </summary>
    <author>
      <name><?php echo $job->getCompany() ?></name>
    </author>
  </entry>
<?php endforeach ?>
```

Vous pouvez utiliser le partial `_list.atom.php` pour simplifier le Template du flux emploi :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.atom.php -->
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Jobeet</title>
  <subtitle>Latest Jobs</subtitle>
  <link href="<?php echo url_for('job', array('sf_format' => 'atom'), true) ?>"
rel="self"/>
  <link href="<?php echo url_for('homepage', true) ?>"/>
  <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
Doctrine_Core::getTable('JobeetJob')->getLatestPost()-
>getDateTimeObject('created_at')->format('U')) ?></updated>
  <author>
    <name>Jobeet</name>
  </author>
  <id><?php echo sha1(url_for('job', array('sf_format' => 'atom'), true)) ?></id>

<?php foreach ($categories as $category): ?>
  <?php include_partial('job/list', array('jobs' => $category-
>getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')))) ?>
<?php endforeach ?>
</feed>
```

Enfin, créez le Template `showSuccess.atom.php` :

```
<!-- apps/frontend/modules/category/templates/showSuccess.atom.php -->
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Jobeet (<?php echo $category ?>)</title>
  <subtitle>Latest Jobs</subtitle>
  <link href="<?php echo url_for('category', array('sf_subject' => $category,
'sf_format' => 'atom'), true) ?>" rel="self" />
  <link href="<?php echo url_for('category', array('sf_subject' => $category), true) ?
>" />
  <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ', $category->getLatestPost()-
>getDateTimeObject('created_at')->format('U')) ?></updated>
  <author>
```

```
    <name>Jobeet</name>
  </author>
  <id><?php echo sha1(url_for('category', array('sf_subject' => $category), true)) ?
></id>

  <?php include_partial('job/list', array('jobs' => $pager->getResults())) ?>
</feed>
```

Quant au flux principal des emplois, nous avons besoin de la date du dernier emploi pour une catégorie :

```
// lib/model/doctrine/JobeetCategory.class.php
class JobeetCategory extends BaseJobeetCategory
{
  public function getLatestPost()
  {
    return $this->getActiveJobs(1)->getFirst();
  }

  // ...
}
```

À demain

Comme avec de nombreuses fonctionnalités de symfony, le support du format natif vou permet d'ajouter des flux à vos sites Web sans effort.

Aujourd'hui, nous avons amélioré l'expérience du demandeur d'emploi. Demain, nous allons voir comment assurer une plus grande exposition aux annonceurs en leur fournissant un service Web.

« Jour 13 : L'utilisateur

Jour 15 : Services Web »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

the SENSIOLABS network

Since 1998, Sensio Labs has been promoting the Open-Source software movement by providing quality web application development, training, consulting.
Sensio Labs also supports several large Open-Source projects.

Open-Source Products

Symfony - MVC framework
Symfony Components
Doctrine - ORM
Swift Mailer - Mailing library
Twig - Template library
Pirum - PEAR channel server

Servic

Training
Guru -
Partner
Books -
Confere
Confere

Practical symfony
Jour 15 : Services Web

Avec l'ajout des flux sur Jobeet, les demandeurs d'emploi peuvent désormais être informé sur les nouveaux emplois en temps réel.

De l'autre côté de la barrière, lorsque vous postez un emploi, vous voudrez avoir la plus grande exposition possible. Si votre emploi est diffusé sur beaucoup de petits sites Web, vous aurez une meilleure chance de trouver la bonne personne. C'est la puissance de la longue traîne. Les affiliés seront en mesure de publier plus tard les offres d'emploi sur leurs sites Web grâce aux services web que nous allons développer aujourd'hui.

Les Affiliés

Les exigences décrites lors du jour 2 :

"Histoires F7 : Un affilié récupère la liste actuelle des emplois actifs"

Les jeux de test

Créons un nouveau fichier fixture pour les affiliés :

```
# data/fixtures/affiliates.yml
JobeetAffiliate:
  sensio_labs:
    url:        http://www.sensio-labs.com/
    email:      fabien.potencier@example.com
    is_active: true
    token:      sensio_labs
    JobeetCategories: [programming]

  symfony:
    url:        http://www.symfony-project.org/
    email:      fabien.potencier@example.org
    is_active: false
    token:      symfony
    JobeetCategories: [design, programming]
```

La création d'enregistrements des relations plusieurs-vers-plusieurs est aussi simple que la définition d'un tableau avec la clé qui est le nom de la relation. Le contenu du tableau est les noms des objets tels qu'ils sont définis dans les fichiers fixture. Vous pouvez lier des objets à partir de fichiers différents, mais les noms doivent être définies au préalable.

Dans le fichier fixture, les jetons sont codés en dur pour simplifier le test, mais quand un utilisateur réel demande un compte, le jeton devra être généré :

```
// lib/model/doctrine/JobeetAffiliate.class.php
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function save(Doctrine_Connection $conn = null)
  {
    if (!$this->getToken())
    {
      $this->setToken(sha1($this->getEmail().rand(11111, 99999)));
    }

    return parent::save($conn);
  }

  // ...
}
```

Vous pouvez désormais recharger les données :

```
$ php symfony doctrine:data-load
```

Le web service emploi

Comme toujours, lorsque vous créez une nouvelle ressource, c'est une bonne habitude de définir la première adresse URL :

```
# apps/frontend/config/routing.yml
api_jobs:
  url:     /api/:token/jobs.:sf_format
  class:   sfDoctrineRoute
  param:   { module: api, action: list }
  options: { model: JobeetJob, type: list, method: getForToken }
  requirements:
    sf_format: (?:xml|json|yaml)
```

Pour cette route, la variable spéciale `sf_format` termine l'URL et les valeurs valides sont `xml`, `json` ou `yaml`.

La méthode `getForToken()` est appelée lorsque l'action récupère la collection d'objets liés à la route. Comme nous avons besoin de vérifier que l'affilié est activé, nous avons besoin de modifier le comportement par défaut de la route :

```
// lib/model/doctrine/JobeetJobTable.class.php
class JobeetJobTable extends Doctrine_Table
{
  public function getForToken(array $parameters)
  {
    $affiliate = Doctrine_Core::getTable('JobeetAffiliate') -
>findOneByToken($parameters['token']);
    if (!$affiliate || !$affiliate->getIsActive())
    {
      throw new sfError404Exception(sprintf('Affiliate with token "%s" does not exist
or is not activated.', $parameters['token']));
    }

    return $affiliate->getActiveJobs();
  }

  // ...
}
```

Si le jeton n'existe pas dans la base de données, nous lançons une exception `sfError404Exception`. Cette classe d'exception est ensuite automatiquement converti en une réponse `404`. C'est le moyen le plus simple pour générer une page `404` d'une classe de modèle.

La méthode `getForToken()` utilise une nouvelle méthode nommée `getActiveJobs()` et renvoie la liste des emplois actuellement actifs :

```
// lib/model/doctrine/JobeetAffiliate.class.php
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function getActiveJobs()
  {
    $q = Doctrine_Query::create()
      ->select('j.*')
      ->from('JobeetJob j')
      ->leftJoin('j.JobeetCategory c')
      ->leftJoin('c.JobeetAffiliates a')
      ->where('a.id = ?', $this->getId());

    $q = Doctrine_Core::getTable('JobeetJob')->addActiveJobsQuery($q);

    return $q->execute();
  }

  // ...
}
```

La dernière étape consiste à créer l'action `api` et les Templates. Démarrez le module avec la tâche `generate:module` :

```
$ php symfony generate:module frontend api
```

Comme nous n'utiliserons pas l'action par défaut `index`, vous pouvez la supprimer de la classe action, et retirez le Template associé `indexSucess.php`.

L'action

Tous les formats partagent la même action `list` :

```
// apps/frontend/modules/api/actions/actions.class.php
public function executeList(sfWebRequest $request)
{
  $this->jobs = array();
  foreach ($this->getRoute()->getObjects() as $job)
  {
    $this->jobs[$this->generateUrl('job_show_user', $job, true)] = $job-
>asArray($request->getHost());
  }
}
```

Plutôt que de transmettre un tableau d'objets JobeetJob aux Templates, on passe un tableau de chaînes. Comme nous avons trois Templates différents pour la même action, la logique pour traiter les valeurs a été refactorisée dans la méthode `JobeetJob::asArray()` :

```
// lib/model/doctrine/JobeetJob.class.php
class JobeetJob extends BaseJobeetJob
{
  public function asArray($host)
  {
    return array(
      'category'     => $this->getJobeetCategory()->getName(),
      'type'         => $this->getType(),
      'company'      => $this->getCompany(),
      'logo'         => $this->getLogo() ? 'http://'.$host.'/uploads/jobs/'.$this-
>getLogo() : null,
      'url'          => $this->getUrl(),
      'position'     => $this->getPosition(),
      'location'     => $this->getLocation(),
      'description'  => $this->getDescription(),
      'how_to_apply' => $this->getHowToApply(),
      'expires_at'   => $this->getCreatedAt(),
    );
  }

  // ...
}
```

Le format *xml*

Le support du format `xml` est aussi simple que la création d'un Template :

```
<!-- apps/frontend/modules/api/templates/listSuccess.xml.php -->
<?xml version="1.0" encoding="utf-8"?>
<jobs>
<?php foreach ($jobs as $url => $job): ?>
  <job url="<?php echo $url ?>">
<?php foreach ($job as $key => $value): ?>
    <<?php echo $key ?>><?php echo $value ?></<?php echo $key ?>>
<?php endforeach ?>
  </job>
<?php endforeach ?>
</jobs>
```

Le format *json*

Le support du format JSON est similaire :

```
<!-- apps/frontend/modules/api/templates/listSuccess.json.php -->
[
<?php $nb = count($jobs); $i = 0; foreach ($jobs as $url => $job): ++$i ?>
{
  "url": "<?php echo $url ?>",
<?php $nb1 = count($job); $j = 0; foreach ($job as $key => $value): ++$j ?>
```

```
  "<?php echo $key ?>": <?php echo json_encode($value).($nb1 == $j ? '' : ',') ?>

<?php endforeach ?>
}<?php echo $nb == $i ? '' : ',' ?>

<?php endforeach ?>
]
```

Le format *yaml*

Pour les formats intégrés, symfony fait une certaine configuration en arrière-plan, comme changer le type de contenu ou désactiver la mise en page.

Comme le format YAML n'est pas dans la liste des formats prédéfinis d'une requête, le type de contenu de la réponse peut être modifié et la mise en page désactivée dans l'action :

```
class apiActions extends sfActions
{
  public function executeList(sfWebRequest $request)
  {
    $this->jobs = array();
    foreach ($this->getRoute()->getObjects() as $job)
    {
      $this->jobs[$this->generateUrl('job_show_user', $job, true)] = $job-
>asArray($request->getHost());
    }

    switch ($request->getRequestFormat())
    {
      case 'yaml':
        $this->setLayout(false);
        $this->getResponse()->setContentType('text/yaml');
        break;
    }
  }
}
```

Dans une action, la méthode `setLayout()` modifie le layout par défaut ou le désactive lorsqu'il est à `false`.

Le Template pour YAML se lit comme suit :

```
<!-- apps/frontend/modules/api/templates/listSuccess.yaml.php -->
<?php foreach ($jobs as $url => $job): ?>
-
  url: <?php echo $url ?>

<?php foreach ($job as $key => $value): ?>
  <?php echo $key ?>: <?php echo sfYaml::dump($value) ?>

<?php endforeach ?>
<?php endforeach ?>
```

Si vous essayez d'appeler le service web avec un jeton non-valide, vous aurez une page XML 404 pour le format XML, et une page JSON 404 pour le format JSON. Mais pour le format YAML, symfony ne sait pas quoi rendre.

Lorsque vous créez un format, une erreur de Template personnalisée doit être créé. Le Template sera utilisé pour les pages 404 et pour toutes les autres exceptions.

Comme l'exception devrait être différente dans l'environnement de production et de développement, deux fichiers sont nécessaires (`config/error/exception.yaml.php` pour le débogage et `config/error/error.yaml.php` pour la production) :

```
// config/error/exception.yaml.php
<?php echo sfYaml::dump(array(
  'error'       => array(
    'code'      => $code,
    'message'   => $message,
    'debug'     => array(
      'name'    => $name,
      'message' => $message,
      'traces'  => $traces,
    ),
```

```
)), 4) ?>

// config/error/error.yaml.php
<?php echo sfYaml::dump(array(
  'error'       => array(
    'code'      => $code,
    'message'   => $message,
))) ?>
```

Avant de l'essayer, vous devez créer un layout pour le format YAML :

```
// apps/frontend/templates/layout.yaml.php
<?php echo $sf_content ?>
```

```
~/work/jobeet $ curl http://jobeet.localhost/frontend_dev.php/api/sensio_lab/jobs.yaml
error:
  code: 404
  message: 'Affiliate with token "sensio_lab" does not exist or is not activated.'
  debug:
    name: sfError404Exception
    message: 'Affiliate with token "sensio_lab" does not exist or is not activated.'
    traces:
      - 'at () in SF_ROOT_DIR/lib/model/JobeetJobPeer.php line 12'
      - 'at JobeetJobPeer::getForToken(array(''token'' =&gt; ''sensio_lab'', ''sf_format''
      - 'at call_user_func(array(''JobeetJobPeer'', ''getForToken''), array(''token'' =&gt
      - 'at sfObjectRoute->getObjectForParameters(array(''module'' =&gt; ''api'', ''action
sfPropelPlugin/lib/routing/sfPropelRoute.class.php line 100'
      - 'at sfPropelRoute->getObjectsForParameters(array(''module'' =&gt; ''api'', ''actio
/sfObjectRoute.class.php line 141'
      - 'at sfObjectRoute->getObjects() in SF_ROOT_DIR/apps/frontend/modules/api/actions/a
```

La redéfinition de l'erreur 404 et l'exception des Templates pour des Templates intégrés est aussi simple que de créer un fichier dans le répertoire `config/error/`.

Les Tests du service web

Pour tester le service web, copiez les fixtures de l'affilié de `data/fixtures/` vers le répertoire `test/fixtures/` et remplacez le contenu auto-généré du fichier `apiActionsTest.php` avec le contenu suivant :

```
// test/functional/frontend/apiActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->
  info('1 - Web service security')->

  info('  1.1 - A token is needed to access the service')->
  get('/api/foo/jobs.xml')->
  with('response')->isStatusCode(404)->

  info('  1.2 - An inactive account cannot access the web service')->
  get('/api/symfony/jobs.xml')->
  with('response')->isStatusCode(404)->

  info('2 - The jobs returned are limited to the categories configured for the
affiliate')->
  get('/api/sensio_labs/jobs.xml')->
  with('request')->isFormat('xml')->
  with('response')->begin()->
    isValid()->
    checkElement('job', 32)->
  end()->

  info('3 - The web service supports the JSON format')->
  get('/api/sensio_labs/jobs.json')->
  with('request')->isFormat('json')->
  with('response')->matches('/"category"\: "Programming"/')->

  info('4 - The web service supports the YAML format')->
```

```
  get('/api/sensio_labs/jobs.yaml')->
  with('response')->begin()->
    isHeader('content-type', 'text/yaml; charset=utf-8')->
    matches('/category\: Programming/')->
  end()
;
```

Dans ce test, vous remarquerez trois nouvelles méthodes :

- `isValid()`: Vérifie si oui ou non la réponse XML est bien formée
- `isFormat()` : Elle teste le format d'une requête
- `matches()` : Pour le format non-HTML, elle contrôle que la réponse correspond à l'expression régulière passée en argument

---

La méthode `isValid()` accepte un booléen en guise de premier paramètre qui permet de valider la réponse XML contre sa définition XSD.

$browser->with('response')->isValid(true);

Cette méthode accepte également une chaîne représentant le chemin absolu vers un fichier XSD propriétaire contre lequel la réponse XML doit être validée.

$browser->with('response')->isValid('/path/to/schema/xsd');

Le formulaire de demande d'affiliation

Maintenant que le service web est prêt à être utilisé, nous allons créer le formulaire de création de compte pour les affiliés. Nous allons décrire une fois encore la procédure classique de l'ajout d'une nouvelle fonctionnalité à une application.

Routage

Vous le deviner. La route est la première chose que nous créons :

```
# apps/frontend/config/routing.yml
affiliate:
  class:   sfDoctrineRouteCollection
  options:
    model: JobeetAffiliate
    actions: [new, create]
    object_actions: { wait: get }
```

C'est une collection de route classique de Doctrine avec une nouvelle option de configuration : `actions`. Comme nous n'avons pas besoin des sept actions par défaut définies par la route, l'option `actions` se charge de la route seulement pour les actions `new` et `create`. La route supplémentaire `wait` sera utilisée pour donner certaines informations au sujet de son compte.

Démarrage

La deuxième étape classique est de générer un module :

```
$ php symfony doctrine:generate-module frontend affiliate JobeetAffiliate --non-
verbose-templates
```

Les Templates

La tâche `doctrine:generate-module` génère les sept actions classiques et leurs Templates correspondants. Dans le répertoire `templates/`, supprimer tous les fichiers sauf `_form.php` and `newSuccess.php`. Et pour les fichiers que nous avons gardés, remplacez leur contenu avec le texte suivant:

```
<!-- apps/frontend/modules/affiliate/templates/newSuccess.php -->
<?php use_stylesheet('job.css') ?>

<h1>Become an Affiliate</h1>

<?php include_partial('form', array('form' => $form)) ?>

<!-- apps/frontend/modules/affiliate/templates/_form.php -->
<?php include_stylesheets_for_form($form) ?>
<?php include_javascripts_for_form($form) ?>

<?php echo form_tag_for($form, 'affiliate') ?>
```

```
  <table id="job_form">
    <tfoot>
      <tr>
        <td colspan="2">
          <input type="submit" value="Submit" />
        </td>
      </tr>
    </tfoot>
    <tbody>
      <?php echo $form ?>
    </tbody>
  </table>
</form>
```

Créez le Template `waitSuccess.php` :

```
<!-- apps/frontend/modules/affiliate/templates/waitSuccess.php -->
<h1>Your affiliate account has been created</h1>

<div style="padding: 20px">
  Thank you!
  You will receive an email with your affiliate token
  as soon as your account will be activated.
</div>
```

Enfin, modifiez le lien dans le pied de page pour qu'il pointe vers le module `affiliate` :

```
// apps/frontend/templates/layout.php
<li class="last">
  <a href="<?php echo url_for('affiliate_new') ?>">Become an affiliate</a>
</li>
```

Les actions

Ici encore, comme nous n'utiliserons que le formulaire de création, ouvrez le fichier `actions.class.php` et supprimez toutes les méthodes sauf `executeNew()`, `executeCreate()` et `processForm()`.

Pour l'action `processForm()`, modifiez l'URL de redirection pour l'action `wait` :

```
// apps/frontend/modules/affiliate/actions/actions.class.php
$this->redirect($this->generateUrl('affiliate_wait', $jobeet_affiliate));
```

L'action `wait` est simple car nous n'avons pas besoin de passer quelque chose pour le Template :

```
// apps/frontend/modules/affiliate/actions/actions.class.php
public function executeWait(sfWebRequest $request)
{
}
```

L'affilié ne peut pas choisir son jeton, ni activer son compte sans tarder. Ouvrez le fichier `JobeetAffiliateForm` pour personnaliser le formulaire :

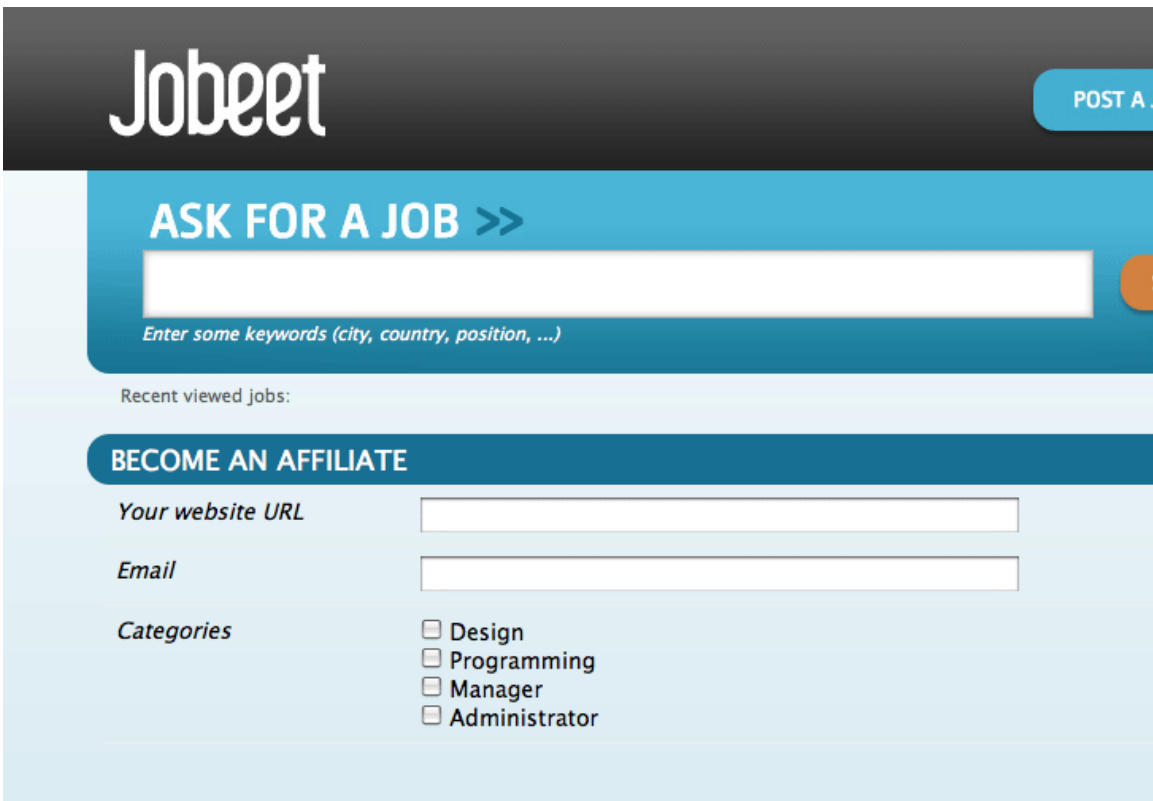
```
// lib/form/doctrine/JobeetAffiliateForm.class.php
class JobeetAffiliateForm extends BaseJobeetAffiliateForm
{
  public function configure()
  {
    $this->useFields(array(
      'url',
      'email',
      'jobeet_categories_list'
    ));
    $this->widgetSchema['jobeet_categories_list']->setOption('expanded', true);
    $this->widgetSchema['jobeet_categories_list']->setLabel('Categories');

    $this->validatorSchema['jobeet_categories_list']->setOption('required', true);

    $this->widgetSchema['url']->setLabel('Your website URL');
    $this->widgetSchema['url']->setAttribute('size', 50);

    $this->widgetSchema['email']->setAttribute('size', 50);
```

```
    $this->validatorSchema['email'] = new sfValidatorEmail(array('required' => true));
  }
}
```

La nouvelle méthode `sfForm::useFields()` permet de spécifier une liste blanche des champs à conserver. Tous les autres champs non mentionnés seront retirés du formulaire.

Le framework de formulaire prend en charge les relations plusieurs-vers-plusieurs comme n'importe quel autre colonne. Par défaut, une telle relation est rendue comme une liste déroulante grâce au widget `sfWidgetFormChoice`. Comme on le voit pendant le jour 10, nous avons changé le tag rendu en utilisant l'option expanded.

Comme les emails et les URL ont tendance à être très supérieurs à la taille par défaut d'une balise input, les attributs HTML par défaut peuvent être définis en utilisant la méthode `setAttribute()`.



Les tests

La dernière étape consiste à écrire des tests fonctionnels pour la nouvelle fonctionnalité.

Remplacez les tests générés pour le module `affiliate` par le code suivant :

```
// test/functional/frontend/affiliateActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->
  info('1 - An affiliate can create an account')->

  get('/affiliate/new')->
  click('Submit', array('jobeet_affiliate' => array(
    'url'                          => 'http://www.example.com/',
    'email'                        => 'foo@example.com',
    'jobeet_categories_list'       =>
array(Doctrine_Core::getTable('JobeetCategory')->findOneBySlug('programming')->getId()),
  )))->
  with('response')->isRedirected()->
  followRedirect()->
  with('response')->checkElement('#content h1', 'Your affiliate account has been created')->
```

```
  info('2 - An affiliate must at least select one category')->

  get('/affiliate/new')->
  click('Submit', array('jobeet_affiliate' => array(
    'url'   => 'http://www.example.com/',
    'email' => 'foo@example.com',
  )))->
  with('form')->isError('jobeet_categories_list')
;
```

Le backend des affiliés

Pour le backend, un module `affiliate` doit être créé pour activer les affiliés par l'administrateur :

```
$ php symfony doctrine:generate-admin backend JobeetAffiliate --module=affiliate
```

Pour accéder au nouveau module créé, ajoutez un lien dans le menu principal avec le nombre d'affiliés qui ont besoin d'être activés :

```
<!-- apps/backend/templates/layout.php -->
<li>
  <a href="<?php echo url_for('jobeet_affiliate') ?>">
    Affiliates - <strong><?php echo Doctrine_Core::getTable('JobeetAffiliate')->
countToBeActivated() ?></strong>
  </a>
</li>

// lib/model/doctrine/JobeetAffiliateTable.class.php
class JobeetAffiliateTable extends Doctrine_Table
{
  public function countToBeActivated()
  {
    $q = $this->createQuery('a')
      ->where('a.is_active = ?', 0);

    return $q->count();
  }

  // ...

}
```

Comme la seule action nécessaire dans le backend est d'activer ou de désactiver des comptes, changez la section `config` du générateur par défaut pour simplifier un peu l'interface et ajouter un lien pour activer des comptes directement à partir de la vue de liste :

```
# apps/backend/modules/affiliate/config/generator.yml
config:
  fields:
    is_active: { label: Active? }
  list:
    title:   Affiliate Management
    display: [is_active, url, email, token]
    sort:    [is_active]
    object_actions:
      activate:   ~
      deactivate: ~
    batch_actions:
      activate:   ~
      deactivate: ~
    actions: {}
  filter:
    display: [url, email, is_active]
```

Pour rendre les administrateurs plus productifs, changez les filtres par défaut pour ne montrer que les affiliées a activé :

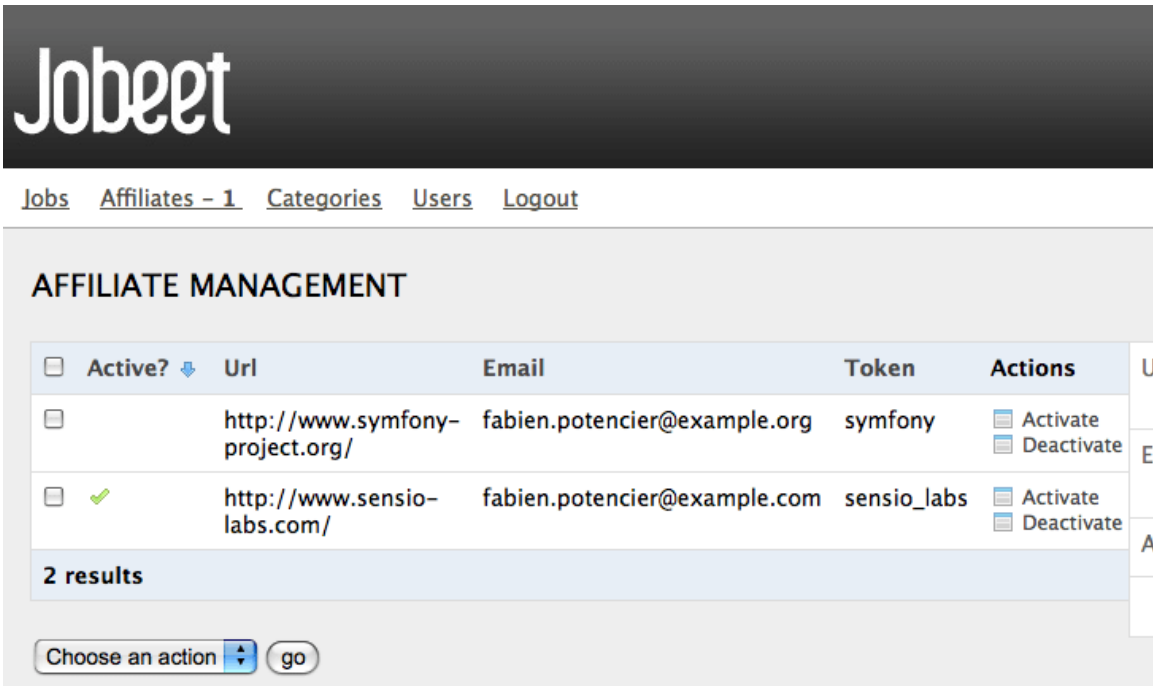
```
// apps/backend/modules/affiliate/lib/affiliateGeneratorConfiguration.class.php
class affiliateGeneratorConfiguration extends BaseAffiliateGeneratorConfiguration
{
  public function getFilterDefaults()
```

```
  {
    return array('is_active' => '0');
  }
}
```

Le seul autre code à écrire, c'est les actions activate et deactivate :

```
// apps/backend/modules/affiliate/actions/actions.class.php
class affiliateActions extends autoAffiliateActions
{
  public function executeListActivate()
  {
    $this->getRoute()->getObject()->activate();

    $this->redirect('jobeet_affiliate');
  }

  public function executeListDeactivate()
  {
    $this->getRoute()->getObject()->deactivate();

    $this->redirect('jobeet_affiliate');
  }

  public function executeBatchActivate(sfWebRequest $request)
  {
    $q = Doctrine_Query::create()
      ->from('JobeetAffiliate a')
      ->whereIn('a.id', $request->getParameter('ids'));

    $affiliates = $q->execute();

    foreach ($affiliates as $affiliate)
    {
      $affiliate->activate();
    }

    $this->redirect('jobeet_affiliate');
  }

  public function executeBatchDeactivate(sfWebRequest $request)
  {
    $q = Doctrine_Query::create()
      ->from('JobeetAffiliate a')
      ->whereIn('a.id', $request->getParameter('ids'));

    $affiliates = $q->execute();

    foreach ($affiliates as $affiliate)
    {
      $affiliate->deactivate();
    }

    $this->redirect('jobeet_affiliate');
  }
}

// lib/model/doctrine/JobeetAffiliate.class.php
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function activate()
  {
    $this->setIsActive(true);

    return $this->save();
  }

  public function deactivate()
  {
    $this->setIsActive(false);

    return $this->save();
```

```
    }

    // ...
}
```



À demain

Merci à l'architecture REST de symfony, il est assez facile à mettre en œuvre des services web pour vos projets. Mais, nous avons écrit le code pour un service web seulement en lecture aujourd'hui, vous avez assez de connaissances symfony pour mettre en œuvre un service web de lecture-écriture.

La mise en œuvre du formulaire de création du compte affilié dans le frontend et son homologue dans le backend était vraiment facile, car vous vous êtes maintenant familiarisé avec le processus d'ajout de nouvelles fonctionnalités à votre projet.

Si vous vous rappelez les exigences du jour 2 :

"L'affilié peut également limiter le nombre d'emplois qui seront retourné et affiner sa requête en spécifiant une catégorie."

La mise en œuvre de cette fonctionnalité est si facile que nous allons vous permettre de le faire ce soir.

Chaque fois qu'un compte d'affilié est activé par l'administrateur, un e-mail doit être envoyé à la filiale pour valider son inscription et lui donner son jeton. L'envoi de courriers électroniques est le sujet dont nous allons parler demain.

« Jour 14 : Les Flux

Jour 16 : L'Envoi d'email »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Since 1998, Sensio Labs has been promoting the Open-Source software movement by providing quality web application development, training, consulting.
Sensio Labs also supports several large Open-Source projects.

Symfony - MVC framework
Symfony Components
Doctrine - ORM
Swift Mailer - Mailing library
Twig - Template library
Pirum - PEAR channel server

Training
Guru -
Partner
Books -
Confere
Confere

Practical symfony
Jour 16 : L'Envoi d'email

Hier, nous avons ajouté à Jobeet un service web en lecture seule. Les affiliés peuvent désormais créer un compte mais il a besoin d'être activé par l'administrateur avant de pouvoir l'utiliser. Pour que l'affilié obtienne son jeton, il nous faut encore mettre en œuvre l'email de notification. C'est ce que nous allons commencer à faire aujourd'hui.

Le framework symfony est livré avec l'une des meilleures solutions d'email PHP : Swift Mailer. Bien sûr, la bibliothèque est entièrement intégré avec symfony, avec quelques fonctionnalités ajoutées à ses fonctionnalités par défaut.

Symfony 1.3/1.4 utilise Swift Mailer version 4.1.

Envoi de simples emails

Commençons par envoyer un simple email pour notifier à l'affiliée lorsque son compte a été confirmé et pour lui donner le jeton de l'affiliée.

Remplacez l'action `activate` avec le code suivant :

```
// apps/backend/modules/affiliate/actions/actions.class.php
class affiliateActions extends autoAffiliateActions
{
  public function executeListActivate()
  {
    $affiliate = $this->getRoute()->getObject();
    $affiliate->activate();

    // send an email to the affiliate
    $message = $this->getMailer()->compose(
      array('jobeet@example.com' => 'Jobeet Bot'),
      $affiliate->getEmail(),
      'Jobeet affiliate token',
      <<<EOF
Your Jobeet affiliate account has been activated.

Your token is {$affiliate->getToken()}.

The Jobeet Bot.
EOF
    );

    $this->getMailer()->send($message);

    $this->redirect('jobeet_affiliate');
  }

  // ...
}
```

Pour que le code fonctionne correctement, vous devez changer l'adresse email `jobeet@example.com` par une réelle.

La gestion des emails dans symfony est centrée autour d'un objet mailer, qui peut être récupéré à partir d'une action avec la méthode `getMailer()`.

La méthode `compose()` prend quatre arguments et retourne un objet de message email :

- l'adresse email de l'expéditeur (`from`);

- l'adresse(s) de(s) destinataire(s) de l'email (`to`);
- l'objet du message;
- le corps du message.

L'envoi du message est donc aussi simple que d'appeler la méthode send `send()` sur l'instance mailer et en passant le message comme un argument. Comme un raccourci, vous pouvez composer et envoyer un email en une seule fois en utilisant la méthode `composeAndSend()`.

Le message email est une instance de la classe `Swift_Message`. Référez vous à la documentation officielle de Swift Mailer pour ven savoir plus sur cet objet, et la façon de faire des choses plus avancées comme joindre des fichiers.

Configuration

Par défaut, la méthode `send()` essaie d'utiliser un serveur SMTP local pour envoyer le message au destinataire. Bien entendu, comme beaucoup de choses dans symfony, ceci est totalement configurable.

Factories

Au cours des jours précédents, nous avons déjà parlé des objets du noyau de symfony comme `user`, `request`, `response` ou `routing`. Ces objets sont automatiquement créés, configurés et gérés par le framework symfony. Ils sont toujours accessibles à partir de l'objet `sfContext`, et comme beaucoup de choses dans le framework, ils sont configurables via un fichier de configuration : `factories.yml`. Ce fichier est configurable par environnement.

Lorsque le `sfContext` initialise les factories du noyau, il lit le fichier `factories.yml` pour passer au constructeur les noms des classes (`class`) et des paramètres (`param`) :

```
response:
  class: sfWebResponse
  param:
    send_http_headers: false
```

Dans l'extrait ci-dessus, pour créer le factory response, symfony instancie un objet `sfWebResponse` et passe l'option `send_http_headers` comme un paramètre.

La classe *sfContext*

L'objet `sfContext` contient des références à des objets du noyau de symfony comme la requête, la réponse, l'utilisateur et ainsi de suite. Comme `sfContext` agit comme un singleton, vous pouvez utiliser l'instruction `sfContext::getInstance()` pour l'obtenir n'importe où et ensuite avoir accès à tous les objets du noyau de symfony :

```
$mailer = sfContext::getInstance()->getMailer();
```

Chaque fois que vous voulez utiliser le `sfContext::getInstance()` dans l'une de vos classes, réfléchissez à deux fois, car elle introduit un couplage fort. Il est toujours mieux de passer l'objet dont vous avez besoin en tant qu'argument.

Vous pouvez même utiliser `sfContext` comme un registre et ajoutez vos propres objets en utilisant les méthodes `set()`. Il prend un nom et un objet comme arguments et la méthode `get()` peut être utilisée plus tard pour récupérer un objet par son nom :

```
sfContext::getInstance()->set('job', $job);
$job = sfContext::getInstance()->get('job');
```

Stratégie d'envoi

Comme beaucoup d'autres objets du noyau de symfony, le logiciel de courrier est un factory. Donc, il est configuré dans le fichier de configuration `factories.yml`. La configuration par défaut se lit comme suit :

```
mailer:
  class: sfMailer
  param:
    logging:           %SF_LOGGING_ENABLED%
    charset:           %SF_CHARSET%
    delivery_strategy: realtime
```

```
    transport:
      class: Swift_SmtpTransport
      param:
        host:       localhost
        port:       25
        encryption: ~
        username:   ~
        password:   ~
```

Lorsque vous créez une nouvelle application, le fichier de configuration locale `factories.yml` remplace la configuration par défaut avec certaines valeurs adaptées pour les environnements env et `test` :

```
test:
  mailer:
    param:
      delivery_strategy: none

dev:
  mailer:
    param:
      delivery_strategy: none
```

Le paramètre `delivery_strategy` explique comment envoyer des emails. Par défaut, symfony vient avec quatre stratégies différentes :

- `realtime`: Les messages sont envoyés en temps réel.
- `single_address`: Les messages sont envoyés à une seule adresse.
- `spool`: Les messages sont stockés dans une file d'attente.
- `none`: Les messages sont simplement ignorés.

Quelle que soit la stratégie, les emails sont toujours journalisés et disponibles dans le panneau "mailer" de la barre d'outils de débogage web.

Transport des emails

Les messages emails sont effectivement envoyés par un transport. Le transport est configuré dans le fichier de configuration `factories.yml` et la configuration par défaut utilise le serveur SMTP de la machine locale :

```
transport:
  class: Swift_SmtpTransport
  param:
    host:       localhost
    port:       25
    encryption: ~
    username:   ~
    password:   ~
```

Swift Mailer st livré avec trois classes de transport différents :

- `Swift_SmtpTransport`: Utilise le serveur SMTP pour envoyer les messages.
- `Swift_SendmailTransport`: Utilise `sendmail` pour envoyer les messages.
- `Swift_MailTransport`: Utilise la fonction native PHP `mail()` pour envoyer les messages.

La section "Transport Types" de la documentation officielle Swift Mailer décrit tout ce que vous devez savoir sur les classes intégrées de transport et leurs différents paramètres.

Test des emails

Maintenant que nous avons vu comment envoyer un email avec le mailer de symfony, écrivons des tests fonctionnels afin de nous assurer du bon déroulement. Par défaut, symfony recense un testeur `mailer` (`sfMailerTester`) pour faciliter les tests de messagerie dans les tests fonctionnels.

Tout d'abord, changez la configuration de l'objet `mailer` pour l'environnement de `test` si le serveur web local n'est pas équipé d'un service SMTP. Nous devons remplacer l'actuelle classe de transport `Swift_SmtpTransport` par la classe `Swift_MailTransport`:

```
# apps/backend/config/factories.yml
test:
```

```
  # ...

  mailer:
    param:
      delivery_strategy: none
      transport:
        class:  Swift_MailTransport
```

Ensuite, ajoutez un nouveau fichier `test/fixtures/administrators.yml` contenant la définition YAML suivante:

```
sfGuardUser:
  admin:
    email_address: admin@example.com
    username: admin
    password: admin
    first_name: Fabien
    last_name: Potencier
    is_super_admin: true
```

Enfin, remplacez le fichier de test fonctionnel `affiliate` de l'application backend avec le code suivant:

```
// test/functional/backend/affiliateActionsTest.php
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
$browser->loadData();

$browser->
  info('1 - Authentication')->
  get('/affiliate')->
  click('Signin', array(
    'signin' => array('username' => 'admin', 'password' => 'admin'),
    array('_with_csrf' => true)
  ))->
  with('response')->isRedirected()->
  followRedirect()->

  info('2 - When validating an affiliate, an email must be sent with its token')->
  click('Activate', array(), array('position' => 1))->
  with('mailer')->begin()->
    checkHeader('Subject', '/Jobeet affiliate token/')->
    checkBody('/Your token is symfony/')->
  end()
;
```

Chaque message envoyé peut être testé à l'aide des méthodes `checkHeader()` et `checkBody()`. Le deuxième argument de `checkHeader()` et le premier argument de `checkBody()` peuvent être un des éléments suivants :

- une chaîne pour vérifier une correspondance exacte;
- une expression régulière pour vérifier la valeur à son encontre;
- une expression régulière négative (une expression régulière commençant avec `!`) pour vérifier que la valeur ne correspond pas.

Par défaut, les contrôles sont faits sur le premier email envoyé. Si plusieurs emails ont été envoyés, vous pouvez choisir celui que vous voulez tester avec la méthode `withMessage()`. La `withMessage()` prend un destinataire pour son premier argument. Elle prend également un second argument pour indiquer quel email vous souhaitez tester si plusieurs ont été envoyés au un même destinataire.

Comme d'autres testeurs intégrés, vous pouvez voir le message brut en appelant la méthode `debug()`.

À demain

Demain, nous allons mettre en œuvre la dernière fonctionnalité manquante du site Jobeet : le

moteur de recherche.

« Jour 15 : Services Web

Jour 17 : La recherche »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 17 : La recherche

Il y a deux jours, nous avons ajouté certains flux pour tenir au courant les utilisateurs de Jobeet pour les nouveaux emplois. Aujourd'hui, nous allons continuer à améliorer l'expérience utilisateur en implémentant la dernière caractéristique principale du site web Jobeet : le moteur de recherche.

La technologie

Avant de plonger la tête la première, parlons un peu de l'histoire de symfony. Nous plaidons pour un grand nombre de bonnes pratiques, comme les tests et la refactorisation, et nous essayons aussi de les appliquer framework lui-même. Par exemple, nous aimons la fameuse devise "Ne pas réinventer la roue". En fait, le framework symfony a commencé sa vie il y a quatre ans comme colle entre les deux logiciels existants Open Source : Mojavi et Propel. Et chaque fois que nous avons besoin d'affronter un nouveau problème, nous cherchons une bibliothèque existante qui fait bien le travail avant de le coder nous mêmes à partir de zéro.

Aujourd'hui, nous voulons ajouter un moteur de recherche à Jobeet, et le Zend Framework fournit une grande bibliothèque, appelée Zend Lucene, qui est un portage du projet bien connu Java Lucene. Au lieu de créer encore un autre moteur de recherche pour Jobeet, ce qui est une tâche complexe, nous allons utiliser Zend Lucene.

Sur la page de documentation de Zend Lucene, la bibliothèque est décrite comme suit:

> ... est un moteur de recherche de contenus principalement textuels écrit entièrement en PHP 5. Comme il stocke ses index sur le système de fichiers et qu'il ne requiert pas de base de données, il peut offrir des fonctionnalités de recherche à presque n'importe quel site écrit en PHP. Zend_Search_Lucene dispose des caractéristiques suivantes :
>
> - Ranked searching - les meilleurs résultats sont retournés en premier.
> - Plusieurs puissants types de requêtes : phrase, booléen, joker (wildcard), proximité, intervalle et bien d'autres.
> - Recherche par champ spécifique (par exemple titre, auteur, contenus)

Ce chapitre n'est pas un tutoriel sur la bibliothèque de Zend Lucene, mais comment l'intégrer dans le site Web Jobeet; ou, plus généralement, comment intégrer les bibliothèques tierces dans un projet symfony. Si vous souhaitez plus d'informations sur cette technologie, référez vous, s'il vous plaît, à la documentation de Lucene Zend.

Installation et configuration du Zend Framework

La bibliothèque Zend Lucene fait partie du Zend Framework. Nous installerons le Zend Framework dans le répertoire `lib/vendor/`, à côté du framework symfony lui-même.

D'abord, téléchargez le Zend Framework et décompressez les fichiers afin d'avoir un répertoire `lib/vendor/Zend/`.

Les explications suivantes ont été testé avec la version 1.9 du Zend Framework.

Vous pouvez nettoyer le répertoire en enlevant tout sauf les fichiers et les répertoires suivants :

- `Exception.php`
- `Loader/`

Autoloader.php
- Search/

Puis, ajoutez le code suivant à la classe `ProjectConfiguration` pour fournir un moyen simple d'enregistrer le chargeur automatique de Zend :

```
// config/ProjectConfiguration.class.php
class ProjectConfiguration extends sfProjectConfiguration
{
  static protected $zendLoaded = false;

  static public function registerZend()
  {
    if (self::$zendLoaded)
    {
      return;
    }


set_include_path(sfConfig::get('sf_lib_dir').'/vendor'.PATH_SEPARATOR.get_include_path(
;
    require_once sfConfig::get('sf_lib_dir').'/vendor/Zend/Loader/Autoloader.php';
    Zend_Loader_Autoloader::getInstance();
    self::$zendLoaded = true;
  }

  // ...
}
```

Indexation

Le moteur de recherche Jobeet devrait être en mesure de restituer tous les emplois correspondants à des mots-clés entrés par l'utilisateur. Avant d'être en mesure de faire quoi que ce soit pour la recherche, un index doit être construit pour les emplois; pour Jobeet, il sera stocké dans le répertoire `data/`.

Zend Lucene fournit deux méthodes pour récupérer un index selon si celle-ci existe déjà ou non. Nous allons créer une méthode helper dans la classe `JobeetJobTable` qui retourne un index existant ou en crée un nouveau pour nous :

```
// lib/model/doctrine/JobeetJobTable.class.php
static public function getLuceneIndex()
{
  ProjectConfiguration::registerZend();

  if (file_exists($index = self::getLuceneIndexFile()))
  {
    return Zend_Search_Lucene::open($index);
  }
  else
  {
    return Zend_Search_Lucene::create($index);
  }
}

static public function getLuceneIndexFile()
{
  return
sfConfig::get('sf_data_dir').'/job.'.sfConfig::get('sf_environment').'.index';
}
```

La méthode *save()*

Chaque fois qu'un emploi est créé, modifié ou supprimé, l'index doit être mis à jour. Modifiez JobeetJob pour mettre à jour l'index à chaque fois qu'un emploi est sérialisé dans la base de données :

```
public function save(Doctrine_Connection $conn = null)
{
  // ...
```

```
  $ret = parent::save($conn);

  $this->updateLuceneIndex();

  return $ret;
}
```

Et créez la méthode updateLuceneIndex() qui fait tout le boulot :

```
// lib/model/doctrine/JobeetJob.class.php
public function updateLuceneIndex()
{
  $index = JobeetJobTable::getLuceneIndex();

  // remove existing entries
  foreach ($index->find('pk:'.$this->getId()) as $hit)
  {
    $index->delete($hit->id);
  }

  // don't index expired and non-activated jobs
  if ($this->isExpired() || !$this->getIsActivated())
  {
    return;
  }

  $doc = new Zend_Search_Lucene_Document();

  // store job primary key to identify it in the search results
  $doc->addField(Zend_Search_Lucene_Field::Keyword('pk', $this->getId()));

  // index job fields
  $doc->addField(Zend_Search_Lucene_Field::UnStored('position', $this->getPosition(),
'utf-8'));
  $doc->addField(Zend_Search_Lucene_Field::UnStored('company', $this->getCompany(),
'utf-8'));
  $doc->addField(Zend_Search_Lucene_Field::UnStored('location', $this->getLocation(),
'utf-8'));
  $doc->addField(Zend_Search_Lucene_Field::UnStored('description', $this-
>getDescription(), 'utf-8'));

  // add job to the index
  $index->addDocument($doc);
  $index->commit();
}
```

Comme Zend Lucene n'est pas en mesure de mettre à jour une entrée existante, elle est d'abord enlevée si le poste existe déjà dans l'index.

L'indexation de l'emploi lui-même est simple : la clé primaire est stockée pour un référencement ultérieur lors de la recherche d'emplois et les colonnes principales (`position`, `company`, `location` et `description`) sont indexeés, mais pas stockeés dans l'index car nous allons utiliser les objets réels pour afficher les résultats.

Transactions de Doctrine

Et si il y a un problème lors de l'indexation d'un emploi ou si l'emploi n'est pas enregistré dans la base de données ? Doctrine et Zend Lucene lèveront une exception. Mais, dans certaines circonstances, nous pourrions avoir un emploi enregistré dans la base de données sans l'indexation correspondante. Pour éviter cela, on peut envelopper les deux mises à jour dans une transaction et faire rollback en cas d'erreur :

```
// lib/model/doctrine/JobeetJob.class.php
public function save(Doctrine_Connection $conn = null)
{
  // ...

  $conn = $conn ? $conn : JobeetJobTable::getConnection();
  $conn->beginTransaction();
  try
  {
    $ret = parent::save($conn);
```

```
    $this->updateLuceneIndex();

    $conn->commit();

    return $ret;
  }
  catch (Exception $e)
  {
    $conn->rollBack();
    throw $e;
  }
}
```

delete()

Nous avons besoin aussi de surcharger la méthode `delete()` pour supprimer l'entrée de l'emploi supprimé de l'index :

```
// lib/model/doctrine/JobeetJob.class.php
public function delete(Doctrine_Connection $conn = null)
{
  $index = JobeetJobTable::getLuceneIndex();

  foreach ($index->find('pk:'.$this->getId()) as $hit)
  {
    $index->delete($hit->id);
  }

  return parent::delete($conn);
}
```

Recherche

Maintenant que nous avons tout en place, vous pouvez recharger les données de test pour les indexer :

```
$ php symfony doctrine:data-load
```

Pour les utilisateurs Unix : comme l'index est modifié à partir de la ligne de commande et aussi à partir du web, vous devez changer en conséquence les droits du répertoire de l'index en fonction de votre configuration : vérifiez que la ligne de commande utilisateur et le serveur web peuvent écrire dans le répertoire de l'index.

Vous pouvez avoir quelques avertissements sur la classe `ZipArchive` si vous n'avez pas l'extension `zip` compilé dans votre PHP. C'est un bug connu de la classe `Zend_Loader`.

L'implémentation de la recherche dans le frontend, c'est du gâteau. Tout d'abord, créer une route :

```
job_search:
  url:   /search
  param: { module: job, action: search }
```

Et l'action correspondante :

```
// apps/frontend/modules/job/actions/actions.class.php
class jobActions extends sfActions
{
  public function executeSearch(sfWebRequest $request)
  {
    $this->forwardUnless($query = $request->getParameter('query'), 'job', 'index');

    $this->jobs = Doctrine_Core::getTable('JobeetJob') ->getForLuceneQuery($query);
  }

  // ...
}
```

La nouvelle méthode `forwardUnless()` redirige l'utilisateur vers l'action `index` du module `job` si la variable `query` de l'URL n'existe pas ou est vide.

Cette méthode n'est en fait qu'un simple alias pour le code suivant:

```
if (!$query = $request->getParameter('query')) { $this->forward('job', 'index'); }
```

Le Template est également assez simple :

```
// apps/frontend/modules/job/templates/searchSuccess.php
<?php use_stylesheet('jobs.css') ?>

<div id="jobs">
  <?php include_partial('job/list', array('jobs' => $jobs)) ?>
</div>
```

La recherche est elle-même déléguée à la méthode`getForLuceneQuery()` :

```
// lib/model/doctrine/JobeetJobTable.class.php
public function getForLuceneQuery($query)
{
  $hits = self::getLuceneIndex()->find($query);

  $pks = array();
  foreach ($hits as $hit)
  {
    $pks[] = $hit->pk;
  }

  if (empty($pks))
  {
    return array();
  }

  $q = $this->createQuery('j')
    ->whereIn('j.id', $pks)
    ->limit(20);

  $q = $this->addActiveJobsQuery($q);

  return $q->execute();
}
```

Après avoir obtenu tous les résultats de l'index de Lucene, nous filtrons les emplois inactifs et limitons le nombre de résultats à 20.

Pour le faire fonctionner, mettez à jour la mise en page :

```
// apps/frontend/templates/layout.php
<h2>Ask for a job</h2>
<form action="<?php echo url_for('job_search') ?>" method="get">
  <input type="text" name="query" value="<?php echo $sf_request->getParameter('query')
?>" id="search_keywords" />
  <input type="submit" value="search" />
  <div class="help">
    Enter some keywords (city, country, position, ...)
  </div>
</form>
```

Zend Lucene définit un langage de requête riche qui prend en charge des opérations comme les booléens, les caractères génériques, la recherche floue, et bien plus encore. Tout est documenté dans le manuel de Zend Lucene

---

Tests unitaires

Quel genre de tests unitaires avons-nous besoin de créer pour tester le moteur de recherche? De toute évidence, nous ne testerons pas la bibliothèque Zend Lucene elle-même, mais son intégration avec la classe `JobeetJob`.

Ajouter les tests suivants à la fin du fichier `JobeetJobTest.php` et n'oubliez pas de mettre à jour le nombre de tests à 7 au début du fichier :

```php
// test/unit/model/JobeetJobTest.php
$t->comment('->getForLuceneQuery()');
$job = create_job(array('position' => 'foobar', 'is_activated' => false));
$job->save();
$jobs = Doctrine_Core::getTable('JobeetJob')->getForLuceneQuery('position:foobar');
$t->is(count($jobs), 0, '::getForLuceneQuery() does not return non activated jobs');

$job = create_job(array('position' => 'foobar', 'is_activated' => true));
$job->save();
$jobs = Doctrine_Core::getTable('JobeetJob')->getForLuceneQuery('position:foobar');
$t->is(count($jobs), 1, '::getForLuceneQuery() returns jobs matching the criteria');
$t->is($jobs[0]->getId(), $job->getId(), '::getForLuceneQuery() returns jobs matching
the criteria');

$job->delete();
$jobs = Doctrine_Core::getTable('JobeetJob')->getForLuceneQuery('position:foobar');
$t->is(count($jobs), 0, '::getForLuceneQuery() does not return deleted jobs');
```

Nous testons un emploi non activé ou une suppression non présente dans les résultats de la recherche, nous vérifions également que les emplois correspondants aux critères donnés s'affichent dans les résultats.

Tâches

Finalement, nous avons besoin de créer une tâche de nettoyage de l'index à partir des vieilles entrées (lorsqu'un emploi prend fin par exemple) et optimiser l'index de temps en temps. Comme nous avons déjà une tâche de nettoyage, nous allons la mettre à jour pour ajouter ces fonctionnalités :

```php
// lib/task/JobeetCleanupTask.class.php
protected function execute($arguments = array(), $options = array())
{
  $databaseManager = new sfDatabaseManager($this->configuration);

  // cleanup Lucene index
  $index = JobeetJobTable::getLuceneIndex();

  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.expires_at < ?', date('Y-m-d'));

  $jobs = $q->execute();
  foreach ($jobs as $job)
  {
    if ($hit = $index->find('pk:'.$job->getId()))
    {
      $index->delete($hit->id);
    }
  }

  $index->optimize();

  $this->logSection('lucene', 'Cleaned up and optimized the job index');

  // Remove stale jobs
  $nb = Doctrine_Core::getTable('JobeetJob')->cleanup($options['days']);

  $this->logSection('doctrine', sprintf('Removed %d stale jobs', $nb));
}
```

La tâche supprime de l'index tous les emplois expirés, puis l'optimise grâce à la méthode `optimize` de Zend Lucene.

À demain

Aujourd'hui, nous avons implémenté un moteur de recherche complet avec de nombreuses fonctionnalités en moins d'une heure. Chaque fois que vous souhaitez ajouter une nouvelle fonctionnalité à vos projets, vérifier qu'elle n'a pas encore été faite ailleurs. Tout d'abord, vérifier si quelque chose n'est pas implémenté nativement dans le framework symfony. Ensuite, vérifiez les plugins de symfony. Et n'oubliez pas de consulter les bibliothèques du Zend Framework et les aussi ezComponent.

Demain, nous emploierons quelques Javascript discrets pour améliorer la réactivité du moteur de recherche en mettant à jour les résultats en temps réel pendant que l'utilisateur tape dans la boîte de recherche. Bien sûr, ce sera l'occasion de parler de la façon d'utiliser AJAX avec symfony.

« Jour 16 : L'Envoi d'email

Jour 18 : AJAX »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

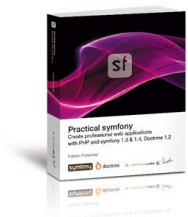
If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 18 : AJAX

Hier, nous avons mis en place un moteur de recherche très puissant pour Jobeet, grâce à la librairie Lucene Zend.

Aujourd'hui, pour renforcer la réactivité du moteur de recherche, nous tirerons profit d'AJAX pour rendre le moteur de recherche plus vivant.

Comme le formulaire doit travailler avec et sans l'activation du Javascript, la fonction de recherche en direct sera mise en œuvre en utilisant du Javascript discret. L'utilisation discrète permet aussi une meilleure séparation des préoccupations dans le code client entre le HTML, les CSS, le JavaScript et les comportements.



Installation de jQuery

Au lieu de réinventer la roue et de gérer les nombreuses différences entre les navigateurs, nous allons utiliser une bibliothèque JavaScript : jQuery. Le framework Symfony est lui-même agnostique et peut fonctionner avec n'importe quelle bibliothèque JavaScript.

Accéder au site jQuery, téléchargez la dernière version et mettez le fichier `.js` sous `web/js/`.

Inclusion de jQuery

Comme nous aurons besoin de jQuery sur toutes les pages, mettez à jour le layout pour l'inclure dans <head>. Faites attention d'insérer la fonction `use_javascript()` avant l'appel de `include_javascripts()` :

```
<!-- apps/frontend/templates/layout.php -->

  <?php use_javascript('jquery-1.2.6.min.js') ?>
  <?php include_javascripts() ?>
</head>
```

Nous aurions pu inclure le fichier jQuery directement avec une balise `<script>`, mais en utilisant le helper `use_javascript()`, cela garantit que le même fichier JavaScript ne sera pas inclus deux fois.

Pour des raisons de performances, vous pouvez aussi déplacer l'appel du helper `include_javascripts()` juste avant la fin de la balise `</body>`.

Ajout des comportements

Mettre en œuvre une recherche en direct signifie que chaque fois que l'utilisateur tape une lettre dans la boîte de recherche, un appel vers le serveur doit être déclenché, le serveur renverra alors les informations nécessaires pour mettre à jour certaines régions de la page sans rafraîchir toute la page.

Au lieu d'ajouter le comportement avec un attribut HTML on*(), le principe essentiel de jQuery est d'ajouter des comportements au DOM après que la page soit complètement chargée. De cette façon, si vous désactivez le support JavaScript dans votre navigateur, aucun comportement n'est enregistré, et le formulaire fonctionne toujours comme avant.

La première étape est d'intercepter chaque fois qu'un utilisateur tape sur une touche dans le champ de recherche :

```
$('#search_keywords').keyup(function(key)
{
  if (this.value.length >= 3 || this.value == '')
  {
    // do something
  }
```

```
});
```

N'ajoutez pas le code pour l'instant, comme nous allons beaucoup le modifier. Le code final JavaScript sera ajoutée au layout dans la section suivante.

Chaque fois que l'utilisateur tape sur une touche, jQuery exécute la fonction anonyme définie dans le code ci-dessus, mais seulement si l'utilisateur a tapé plus de 3 caractères ou s'il a enlevé quelque chose de la balise input.

Faire un appel AJAX pour le serveur est aussi simple que d'utiliser la méthode `load()` sur l'élément du DOM :

```
$('#search_keywords').keyup(function(key)
{
  if (this.value.length >= 3 || this.value == '')
  {
    $('#jobs').load(
      $(this).parents('form').attr('action'), { query: this.value + '*' }
    );
  }
});
```

Pour gérer l'appel AJAX, c'est la même action "normale" qui est appelée. Les changements nécessaires dans l'action se feront dans la prochaine section.

Enfin et surtout, si JavaScript est activé, nous voulons supprimer le bouton de recherche :

```
$('.search input[type="submit"]').hide();
```

Remontée de l'information à l'utilisateur

Chaque fois que vous effectuez un appel AJAX, la page ne sera pas mise à jour immédiatement. Le navigateur attendra la réponse du serveur avant d'actualiser la page. Dans l'intervalle, vous devez fournir la remontée de l'information visuelle à l'utilisateur pour l'informer que quelque chose se passe.

Une convention est d'afficher une icône du chargeur lors de l'appel AJAX. Mettez à jour le layout pour ajouter l'image du chargeur et cachez le par défaut :

```
<!-- apps/frontend/templates/layout.php -->
<div class="search">
  <h2>Ask for a job</h2>
  <form action="<?php echo url_for('job_search') ?>" method="get">
    <input type="text" name="query" value="<?php echo $sf_request-
>getParameter('query') ?>" id="search_keywords" />
    <input type="submit" value="search" />
    <img id="loader" src="/images/loader.gif" style="vertical-align: middle; display:
none" />
    <div class="help">
      Enter some keywords (city, country, position, ...)
    </div>
  </form>
</div>
```

Le chargeur par défaut est optimisé pour le layout actuel de Jobeet. Si vous voulez créer le vôtre, vous trouverez une foule de services gratuits en ligne comme http://www.ajaxload.info/.

Maintenant que vous avez toutes les pièces nécessaires pour faire fonctionner le HTML, créez un fichier `search.js` qui contient le code JavaScript que nous avons écrit à ce jour :

```
// web/js/search.js
$(document).ready(function()
{
  $('.search input[type="submit"]').hide();

  $('#search_keywords').keyup(function(key)
  {
    if (this.value.length >= 3 || this.value == '')
    {
```

```
      $('#loader').show();
      $('#jobs').load(
        $(this).parents('form').attr('action'),
        { query: this.value + '*' },
        function() { $('#loader').hide(); }
      );
    }
  });
});
```

Vous devez également mettre à jour le layout pour inclure ce nouveau fichier :

```
<!-- apps/frontend/templates/layout.php -->
<?php use_javascript('search.js') ?>
```

JavaScript comme une action

Bien que le JavaScript que nous avons écrit pour le moteur de recherche soit statique, parfois, vous avez besoin d'appeler un peu de code PHP (pour utiliser le helper `url_for()` par exemple).

JavaScript est juste un autre format comme le HTML, et comme cela a été vu il y a quelques jours, symfony rend la gestion du format assez facile. Comme le fichier JavaScript contiendra le comportement d'une page, vous pouvez même avoir la même URL que la page du fichier JavaScript, mais se terminant par `.js`. Par exemple, si vous souhaitez créer un fichier pour le comportement du moteur de recherche, vous pouvez modifier la route `job_search` comme suit et créer un Template `searchSuccess.js.php` :

```
job_search:
  url:    /search.:sf_format
  param: { module: job, action: search, sf_format: html }
  requirements:
    sf_format: (?:html|js)
```

AJAX dans une action

Si JavaScript est activé, jQuery interceptera toutes les touches tapées dans le champ de recherche et appellera l'action `search`. Sinon, la même action même `search` est également appelée lorsque l'utilisateur sousmet le formulaire en appuyant sur la touche "entrée" ou en cliquant sur le bouton "recherche".

Ainsi, l'action `search` doit maintenant déterminer si l'appel se fait via AJAX ou pas. Chaque fois qu'une requête est faite avec un appel AJAX, la méthode `isXmlHttpRequest()` de l'objet de requête retourne `true`.

La méthode `isXmlHttpRequest()` fonctionne avec toutes les grandes bibliothèques JavaScript comme Prototype, Mootools ou JQuery.

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeSearch(sfWebRequest $request)
{
  $this->forwardUnless($query = $request->getParameter('query'), 'job', 'index');

  $this->jobs = Doctrine_Core::getTable('JobeetJob')->getForLuceneQuery($query);

  if ($request->isXmlHttpRequest())
  {
    return $this->renderPartial('job/list', array('jobs' => $this->jobs));
  }
}
```

Comme jQuery ne rechargera pas la page mais ne fera que remplacer l'élément du DOM `#jobs` avec le contenu de la réponse, la page ne devrait pas être décoré par le layout. Comme il s'agit d'un besoin commun, le layout est désactivé par défaut quand une requête AJAX entre en jeu.

En outre, au lieu de retourner le Template complet, il suffit de renvoyer le contenu du partial `job/list`. La méthode `renderPartial()` utilisée dans l'action renvoie le partial comme réponse au lieu de l'intégralité du Template.

Si l'utilisateur supprime tous les caractères dans le champ de recherche, ou si la recherche ne

retourne aucun résultat, nous avons besoin d'afficher un message au lieu d'une page blanche. Nous allons utiliser la méthode `renderText()` pour rendre une chaîne de test simple :

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeSearch(sfWebRequest $request)
{
  $this->forwardUnless($query = $request->getParameter('query'), 'job', 'index');

  $this->jobs = Doctrine_Core::getTable('JobeetJob')->getForLuceneQuery($query);

  if ($request->isXmlHttpRequest())
  {
    if ('*' == $query || !$this->jobs)
    {
      return $this->renderText('No results.');
    }

    return $this->renderPartial('job/list', array('jobs' => $this->jobs));
  }
}
```

Vous pouvez également retourner un Component dans une action en utilisant la méthode `renderComponent()`.

Test d'AJAX

Comme le navigateur symfony ne peut pas simuler du JavaScript, vous avez besoin de l'aider lors des tests des appels AJAX. Cela signifie principalement que vous devez ajouter manuellement l'entête que jQuery et toutes les autres grandes bibliothèques JavaScript envoient avec la requête :

```
// test/functional/frontend/jobActionsTest.php
$browser->setHttpHeader('X_REQUESTED_WITH', 'XMLHttpRequest');
$browser->
  info('5 - Live search')->

  get('/search?query=sens*')->
  with('response')->begin()->
    checkElement('table tr', 2)->
  end()
;
```

La méthode `setHttpHeader()` définit une entête HTTP pour la très prochaine requête faite avec le navigateur.

À demain

Hier, nous avons utilisé la bibliothèque Zend Lucene pour mettre en œuvre le moteur de recherche. Aujourd'hui, nous avons utilisé jQuery pour le rendre plus réactif. Le framework symfony fournit tous les outils essentiels pour construire des applications MVC avec aisance et joue aussi bien avec les autres composants. Comme toujours, essayez d'utiliser le meilleur outil pour le travail.

Demain, nous allons voir comment internationaliser le site Jobeet.

« Jour 17 : La recherche

Jour 19 : Internationalisation et régionalisation »

the SENSIOLABS network

Since 1998, Sensio Labs has been promoting the Open-Source software movement by providing quality web application development, training, consulting.
Sensio Labs also supports several large Open-Source projects.

Open-Source Products

Symfony - MVC framework
Symfony Components
Doctrine - ORM
Swift Mailer - Mailing library
Twig - Template library
Pirum - PEAR channel server

Servic

Training
Guru -
Partner
Books -
Confere
Confere

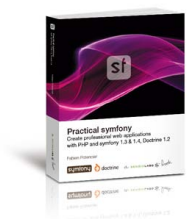
Practical symfony
Jour 19 : Internationalisation et régionalisation

Hier, nous avons terminé la fonctionnalité du moteur de recherche en la rendant encore plus fun avec l'ajout de quelques AJAX de qualité.

Aujourd'hui, nous allons parler de l'**internationalisation** (ou i18n) de Jobeet et la **régionalisation** (ou l10n).

Extrait de Wikipedia :

> L'**internationalisation** est le processus de conception d'un logiciel afin qu'il puisse être adapté aux différentes langues et régions sans modifications techniques.
>
> La **régionalisation** est le processus d'adaptation des logiciels à une région spécifique ou à une langue en y ajoutant des éléments spécifiques locaux et la traduction du texte.



Comme toujours, le framework symfony n'a pas réinventé la roue, son support sur i18n et sur l10n est basé sur le standard ICU.

User

Aucune internationalisation n'est possible sans utilisateur. Quand votre site Web est disponible dans plusieurs langues ou pour différentes régions du monde, l'utilisateur est responsable de choisir celle qui lui convient le mieux.

Nous avons déjà parlé de la classe User de symfony pendant la journée 13.

La Culture de l'utilisateur

Les caractéristiques i18n et l10n de symfony sont basées sur la **culture de l'utilisateur**. La culture est la combinaison de la langue et du pays de l'utilisateur. Par exemple, la culture pour un utilisateur qui parle français est `fr` et la culture pour un utilisateur de la France est `fr_FR`.

Vous pouvez gérer la culture de l'utilisateur en appelant les méthodes `setCulture()` et `getCulture()` sur l'objet User :

```
// in an action
$this->getUser()->setCulture('fr_BE');
echo $this->getUser()->getCulture();
```

La langue est codée par deux caractères minuscules, selon la norme ISO 639-1 et le pays est codé par deux caractères en majuscules, selon la norme ISO 3166-1.

La culture préférée

Par défaut, la culture de l'utilisateur est celle configurée dans le fichier de configuration `settings.yml` :

```
# apps/frontend/config/settings.yml
all:
  .settings:
    default_culture: it_IT
```

Comme la culture est géré par l'objet User, il est stocké dans la session utilisateur. Au cours du développement, si vous changez la culture par défaut, vous devrez effacer le cookie de votre session pour que le nouveau paramètre ait une influence dans votre navigateur.

Lorsqu'un utilisateur démarre une session sur le site Jobeet, nous pouvons également

déterminer la meilleure culture, sur la base des informations fournies par le `Accept-Language` de l'entête HTTP.

La méthode `getLanguages()` de l'objet de requête renvoie un tableau des langues acceptées par l'utilisateur actuel, triées par ordre de préférence :

```
// in an action
$languages = $request->getLanguages();
```

Mais la plupart du temps, votre site ne sera pas disponible dans les 136 langues majeures du monde. La méthode `getPreferredCulture()` retourne le meilleur langage en comparant les langues préférées de l'utilisateur et les langues prises en charge par votre site web :

```
// in an action
$language = $request->getPreferredCulture(array('en', 'fr'));
```

Dans l'appel précédent, la langue retournée sera anglais ou français selon les langues préférées de l'utilisateur, ou en anglais (la première langue dans le tableau) si aucune ne correspond.

La culture dans l'URL

Le site Web Jobeet sera disponible en anglais et en français. Comme une URL ne peut que représenter une ressource unique, la culture doit être incorporée dans l'URL. Pour ce faire, ouvrez le fichier `routing.yml`, et ajoutez la variable spéciale `:sf_culture` pour toutes les routes sauf pour `api_jobs` et homepage. Pour les routes simples, ajoutez `/:sf_culture` devant `url`. Pour les collections de routes, ajoutez une option `prefix_path` qui commence avec `/:sf_culture`.

```
# apps/frontend/config/routing.yml
affiliate:
  class: sfDoctrineRouteCollection
  options:
    model:          JobeetAffiliate
    actions:        [new, create]
    object_actions: { wait: get }
    prefix_path:    /:sf_culture/affiliate

category:
  url:     /:sf_culture/category/:slug.:sf_format
  class:   sfDoctrineRoute
  param:   { module: category, action: show, sf_format: html }
  options: { model: JobeetCategory, type: object }
  requirements:
    sf_format: (?:html|atom)

job_search:
  url:   /:sf_culture/search
  param: { module: job, action: search }

job:
  class: sfDoctrineRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: put, extend: put }
    prefix_path:    /:sf_culture/job
  requirements:
    token: \w+

job_show_user:
  url:     /:sf_culture/job/:company_slug/:location_slug/:id/:position_slug
  class:   sfDoctrineRoute
  options:
    model: JobeetJob
    type: object
    method_for_query: retrieveActiveJob
  param:   { module: job, action: show }
  requirements:
    id:        \d+
    sf_method: get
```

Lorsque la variable `sf_culture` est utilisée dans une route, symfony utilisera automatiquement

sa valeur pour changer la culture de l'utilisateur.

Comme nous avons besoin d'autant de pages d'accueil que de langue supportées (`/en/`, `/fr/`, ...), la page d'accueil par défaut (`/`) doit rediriger vers celle appropriée, conformément à la culture de l'utilisateur. Mais si l'utilisateur n'a pas encore de culture, parce qu'il agit pour la première fois sur Jobeet, la culture privilégiée sera choisie pour lui.

D'abord, ajoutez la méthode `isFirstRequest()` à `myUser`. Elle retourne `true` seulement pour la première requête d'une session utilisateur :

```
// apps/frontend/lib/myUser.class.php
public function isFirstRequest($boolean = null)
{
  if (is_null($boolean))
  {
    return $this->getAttribute('first_request', true);
  }

  $this->setAttribute('first_request', $boolean);
}
```

Ajoutez la route `localized_homepage` :

```
# apps/frontend/config/routing.yml
localized_homepage:
  url:   /:sf_culture/
  param: { module: job, action: index }
  requirements:
    sf_culture: (?:fr|en)
```

Modifiez l'action `index` du module `job` pour implémenter la logique afin de rediriger l'utilisateur vers la «meilleure» page d'accueil lors de la première requête d'une session:

```
// apps/frontend/modules/job/actions/actions.class.php
public function executeIndex(sfWebRequest $request)
{
  if (!$request->getParameter('sf_culture'))
  {
    if ($this->getUser()->isFirstRequest())
    {
      $culture = $request->getPreferredCulture(array('en', 'fr'));
      $this->getUser()->setCulture($culture);
      $this->getUser()->isFirstRequest(false);
    }
    else
    {
      $culture = $this->getUser()->getCulture();
    }

    $this->redirect('localized_homepage');
  }

  $this->categories = Doctrine_Core::getTable('JobeetCategory')->getWithJobs();
}
```

Si la variable `sf_culture` n'est pas présente dans la requête, cela signifie que l'utilisateur est venu sur l'URL `/`. Si tel est le cas et que la session est nouvelle, la culture préférée est utilisée comme culture de l'utilisateur. Sinon, la culture actuelle de l'utilisateur est utilisée.

La dernière étape consiste à rediriger l'utilisateur vers l'URL `localized_homepage`. Notez que la variable `sf_culture` n'a pas été passée dans l'appel de redirection puisque symfony l'ajoute automatiquement pour vous.

Maintenant, si vous essayez d'aller à l'URL `/it/`, symfony va retourner une erreur 404 car nous avons limité la variable `sf_culture` en ou `fr`. Ajouter cette exigence à toutes les routes qui intègrent la culture :

```
requirements:
  sf_culture: (?:fr|en)
```

Culture Testing

Il est temps de tester notre implémentation. Mais avant d'ajouter plus de tests, nous avons besoin de corriger des objets existants. Comme toutes les URL ont changé, modifiez tous les

fichiers de test fonctionnel dans `test/functional/frontend/` et ajoutez `/en` devant toutes les URLs. N'oubliez pas de changer également les URL dans le fichier `lib/test/JobeetTestFunctional.class.php`. Lancez la suite de test pour vérifier que vous avez correctement corrigé les tests :

```
$ php symfony test:functional frontend
```

Le testeur de User donne une méthode `isCulture()` qui teste la culture de l'utilisateur actuel. Ouvrez le fichier `jobActionsTest` et ajoutez les tests suivants :

```
// test/functional/frontend/jobActionsTest.php
$browser->setHttpHeader('ACCEPT_LANGUAGE', 'fr_FR,fr,en;q=0.7');
$browser->
  info('6 - User culture')->

  restart()->

  info('  6.1 - For the first request, symfony guesses the best culture')->
  get('/')->
  with('response')->isRedirected()->
  followRedirect()->
  with('user')->isCulture('fr')->

  info('  6.2 - Available cultures are en and fr')->
  get('/it/')->
  with('response')->isStatusCode(404)
;

$browser->setHttpHeader('ACCEPT_LANGUAGE', 'en,fr;q=0.7');
$browser->
  info('  6.3 - The culture guessing is only for the first request')->

  get('/')->
  with('response')->isRedirected()->
  followRedirect()->
  with('user')->isCulture('fr')
;
```

Changer de langue

Pour l'utilisateur qui veut changer la culture, un formulaire linguistique doit être ajoutée dans le layout. Le framework de formulaire ne prévoit pas une tel formulaire, mais comme le besoin est assez fréquent pour des sites web internationalisé, l'équipe de symfony maintient le sfFormExtraPlugin, qui contient les validateurs, les widgets et les formulaires qui ne peuvent pas être inclus dans le package symfony principal car ils sont trop spécifiques ou qu'ils ont des dépendances externes mais ils sont néanmoins très utiles.

Installez le plugin avec la tâche `plugin:install` :

```
$ php symfony plugin:install sfFormExtraPlugin
```

Ou bien depuis Subversion avec la commande suivante:

```
    $  svn co http://svn.symfony-project.org/plugins/sfFormExtraPlugin/branches/1.3/
plugins/sfFormExtraPlugin
```

Afin que les classes du plugin puissent être chargées par symfony, le plugin `sfFormExtraPlugin` doit être activé dans le fichier `config/ProjectConfiguration.class.php` comme le montre le code ci-dessous:

```
// config/ProjectConfiguration.class.php
public function setup()
{
  $this->enablePlugins(array(
    'sfDoctrinePlugin',
    'sfDoctrineGuardPlugin',
    'sfFormExtraPlugin'
  ));
}
```

Le `sfFormExtraPlugin` contient des widgets qui nécessitent des dépendances externes,

comme les bibliothèques JavaScript. Vous trouverez un widget pour la sélection de date, un pour un éditeur WYSIWYG et d'autres encore. Prenez le temps de lire la documentation où vous trouverez une foule de trucs utiles.

Le plugin `sfFormExtraPlugin` offre un formulaire `sfFormLanguage` pour gérer la sélection de la langue. L'ajout du formulaire linguistique peut être fait dans le layout comme ceci :

Le code ci-dessous n'est pas destiné à être mis en œuvre. Il est là pour vous montrer comment vous pourriez être tenté de mettre en œuvre quelque chose d'une mauvaise manière. Nous allons continuer à vous montrer comment l'implémenter correctement en utilisant symfony.

```
// apps/frontend/templates/layout.php
<div id="footer">
  <div class="content">
    <!-- footer content -->

    <?php $form = new sfFormLanguage(
      $sf_user,
      array('languages' => array('en', 'fr'))
      )
    ?>
    <form action="<?php echo url_for('change_language') ?>">
      <?php echo $form ?><input type="submit" value="ok" />
    </form>
  </div>
</div>
```

Repérez vous un problème ? À droite, la création d'un objet de formulaire n'appartient pas à la couche de la Vue. Il doit être créé à partir d'une action. Mais, comme le code est dans le layout, le formulaire doit être créé pour chaque action, ce qui est loin d'être pratique. Dans de tels cas, vous devez utiliser un **component**. Un component est comme un partial, mais avec du code qui s'y rattachent. Considérez cela comme une action légère.

L'inclusion d'un component à partir d'un Template peut être fait en utilisant le Helper `include_component()` :

```
// apps/frontend/templates/layout.php
<div id="footer">
  <div class="content">
    <!-- footer content -->

    <?php include_component('language', 'language') ?>
  </div>
</div>
```

Le helper prend le module et l'action comme arguments. Le troisième argument peut être utilisé pour passer des paramètres au component.

Créez un module `language` pour accueillir le component et l'action qui va effectivement changer la langue de l'utilisateur :

```
$ php symfony generate:module frontend language
```

Les components sont à définir dans le fichier `actions/components.class.php`.

Créer ce fichier maintenant :

```
// apps/frontend/modules/language/actions/components.class.php
class languageComponents extends sfComponents
{
  public function executeLanguage(sfWebRequest $request)
  {
    $this->form = new sfFormLanguage(
      $this->getUser(),
      array('languages' => array('en', 'fr'))
    );
  }
}
```

Comme vous pouvez le voir, une classe components est très similaire à la classe des actions.

Le Template pour un component utilise les mêmes conventions de nommage qu'un partial : un trait de soulignement (_), suivi par le nom du component :

```
// apps/frontend/modules/language/templates/_language.php
<form action="<?php echo url_for('change_language') ?>">
  <?php echo $form ?><input type="submit" value="ok" />
</form>
```

Puisque le plugin ne prévoit pas l'action qui change effectivement la culture des utilisateurs, modifiez le fichier `routing.yml` pour créer la route de `change_language` :

```
# apps/frontend/config/routing.yml
change_language:
  url:   /change_language
  param: { module: language, action: changeLanguage }
```

Et créez l'action correspondante :

```
// apps/frontend/modules/language/actions/actions.class.php
class languageActions extends sfActions
{
  public function executeChangeLanguage(sfWebRequest $request)
  {
    $form = new sfFormLanguage(
      $this->getUser(),
      array('languages' => array('en', 'fr'))
    );

    $form->process($request);

    return $this->redirect('localized_homepage');
  }
}
```

La méthode `process()` de `sfFormLanguage` prend soin de changer la culture de l'utilisateur, basé sur la soumission du formulaire de l'utilisateur.

Internationalisation

Langues, Jeu de caractère et Encodage

Plusieurs langues ont des jeux de caractères différents. La langue anglaise est la plus simple car elle n'utilise que les caractères ASCII, la langue française est un peu plus complexe avec des caractères accentués comme "é" et les langues comme le russe, le chinois ou l'arabe sont beaucoup plus complexes car tous leurs caractères sont en dehors de la plage ASCII. Ces langues sont définies avec des jeux de caractères totalement différents.

Lorsqu'il s'agit de données internationalisées, il est préférable d'utiliser la norme unicode. L'idée derrière unicode est d'établir un ensemble universel de caractères qui contient tous les caractères de toutes les langues. Le problème avec unicode est qu'un seul caractère peut être représenté avec pas moins de 21 octets. Par conséquent, pour le web, nous utilisons UTF-8, qui fait correspondre les points de code unicode à des séquences de longueur variable d'octets. En UTF-8, les langues les plus utilisés ont leurs caractères codés avec moins de 3 octets.

UTF-8 est le codage par défaut utilisé par symfony, et il est défini dans le fichier de configuration `settings.yml` :

```
# apps/frontend/config/settings.yml
all:
  .settings:
    charset: utf-8
```

Aussi, pour activer la couche d'internationalisation de symfony, vous devez définir le paramètre `i18n` à `true` dans `settings.yml` :

```
# apps/frontend/config/settings.yml
all:
  .settings:
    i18n: true
```

Templates

Un site web internationalisé signifie que l'interface utilisateur est traduite en plusieurs langues.

Dans un Template, toutes les chaînes qui dépendent de la langue doivent être entourées du helper `__()` (remarquez qu'il y a deux caractères de soulignement).

Le helper `__()` fait parti du groupe d'helper `I18N`, qui contient des helpers qui facilitent la gestion i18n dans les Templates. Comme ce groupe de helper n'est pas chargé par défaut, vous devez soit l'ajouter manuellement dans chaque Template avec `use_helper('I18N')` comme nous l'avons fait pour le groupe d'helper `Text`, ou le charger globallement en l'ajoutant au paramètre `standard_helpers` :

```
# apps/frontend/config/settings.yml
all:
  .settings:
    standard_helpers: [Partial, Cache, I18N]
```

Voici comment utiliser le helper `__()` pour le pied de page de Jobeet :

```
// apps/frontend/templates/layout.php
<div id="footer">
  <div class="content">
    <span class="symfony">
      <img src="/images/jobeet-mini.png" />
      powered by <a href="http://www.symfony-project.org/">
      <img src="/images/symfony.gif" alt="symfony framework" /></a>
    </span>
    <ul>
      <li>
        <a href=""><?php echo __('About Jobeet') ?></a>
      </li>
      <li class="feed">
        <?php echo link_to(__('Full feed'), 'job', array('sf_format' => 'atom')) ?>
      </li>
      <li>
        <a href=""><?php echo __('Jobeet API') ?></a>
      </li>
      <li class="last">
        <?php echo link_to(__('Become an affiliate'), 'affiliate_new') ?>
      </li>
    </ul>
    <?php include_component('language', 'language') ?>
  </div>
</div>
```

Le helper `__()` peut prendre la chaîne de la langue par défaut ou vous pouvez également utiliser un identificateur unique pour chaque chaîne. C'est juste une question de goût. Pour Jobeet, nous allons utiliser la première stratégie ainsi les Teplates seront plus lisibles.

Lorsque symfony rend un Template, chaque fois le helper `__()` est appelé, symfony regarde pour une traduction la culture de l'utilisateur actuel. Si une traduction est trouvée, elle est utilisée, sinon le premier argument est retourné comme une valeur de repli.

Toutes les traductions sont stockées dans un catalogue. Le framework i18n fournit un grand nombre de stratégies différentes pour stocker les traductions. Nous allons utiliser le format "XLIFF" qui est une norme et qui est la plus souple. C'est également le stockage utilisé pour l'admin generator et la plupart des plugins de symfony.

Il existe d'autres catalogues de stockage comme `gettext`, `MySQL` et `SQLite`. Comme toujours, jetez un oeil à l'API i18n pour plus de détails.

i18n:extract

Au lieu de créer le fichier du catalogue à la main, utilisez la tâche intégrée `i18n:extract` :

```
$ php symfony i18n:extract frontend fr --auto-save
```

La tâche `i18n:extract` trouve toutes les chaînes qui doivent être traduits en `fr` dans l'application frontend et crée ou met à jour le catalogue correspondant. L'option `--auto-save` enregistre les nouvelles chaînes dans le catalogue. Vous pouvez également utiliser l'option `--auto-delete` pour supprimer automatiquement les chaînes qui n'existent plus.

Dans notre cas, il remplit le fichier que nous avons créé :

```
<!-- apps/frontend/i18n/fr/messages.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xliff PUBLIC "-//XLIFF//DTD XLIFF//EN"
  "http://www.oasis-open.org/committees/xliff/documents/xliff.dtd">
<xliff version="1.0">
  <file source-language="EN" target-language="fr" datatype="plaintext"
      original="messages" date="2008-12-14T12:11:22Z"
      product-name="messages">
    <header/>
    <body>
      <trans-unit id="1">
        <source>About Jobeet</source>
        <target/>
      </trans-unit>
      <trans-unit id="2">
        <source>Feed</source>
        <target/>
      </trans-unit>
      <trans-unit id="3">
        <source>Jobeet API</source>
        <target/>
      </trans-unit>
      <trans-unit id="4">
        <source>Become an affiliate</source>
        <target/>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Chaque traduction est géré par une balise `trans-unit` qui a un attribut `id` unique. Vous pouvez maintenant éditer ce fichier et ajouter des traductions pour la langue française :

```
<!-- apps/frontend/i18n/fr/messages.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xliff PUBLIC "-//XLIFF//DTD XLIFF//EN"
  "http://www.oasis-open.org/committees/xliff/documents/xliff.dtd">
<xliff version="1.0">
  <file source-language="EN" target-language="fr" datatype="plaintext"
      original="messages" date="2008-12-14T12:11:22Z"
      product-name="messages">
    <header/>
    <body>
      <trans-unit id="1">
        <source>About Jobeet</source>
        <target>A propos de Jobeet</target>
      </trans-unit>
      <trans-unit id="2">
        <source>Feed</source>
        <target>Fil RSS</target>
      </trans-unit>
      <trans-unit id="3">
        <source>Jobeet API</source>
        <target>API Jobeet</target>
      </trans-unit>
      <trans-unit id="4">
```

```
        <source>Become an affiliate</source>
        <target>Devenir un affilié</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Comme XLIFF est un format standard, de nombreux outils existent pour faciliter le processus de traduction. Open Language Tools est un projet Open-Source en Java avec un éditeur XLIFF intégrée.

Comme XLIFF est un format basé sur un fichier, les mêmes règles de priorité et de fusion qui existent pour les autres fichiers de configuration de symfony sont également applicables. Les fichiers i18n peuvent exister dans un projet, une application ou un module, et les traductions des fichiers les plus spécifiques substituent ceux des principaux.

Traductions avec arguments

Le principe essentiel de l'internationalisation est de traduire des phrases entières. Mais certaines phrases intègrent des valeurs dynamiques. Dans Jobeet, c'est le cas sur la page d'accueil pour le lien du "more..." :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<div class="more_jobs">
  and <?php echo link_to($count, 'category', $category) ?> more...
</div>
```

Le nombre d'emplois est une variable qui doit être remplacée par un espace réservé pour la traduction :

```
<!-- apps/frontend/modules/job/templates/indexSuccess.php -->
<div class="more_jobs">
  <?php echo __('and %count% more...', array('%count%' => link_to($count, 'category',
$category))) ?>
</div>
```

La chaîne à traduire est maintenant "and %count% more...", et l'espace réservé %count% sera remplacé par le nombre réel lors de l'exécution, grâce à la valeur donnée comme deuxième argument du helper `__()`.

Ajoutez la nouvelle chaîne manuellement en insérant une balise `trans-unit` dans le fichier `messages.xml`, ou utilisez la tâche `i18n:extract` pour mettre à jour le fichier automatiquement :

```
$ php symfony i18n:extract frontend fr --auto-save
```

Après l'exécution de la tâche, ouvrez le fichier XLIFF pour ajouter la traduction française :

```
<trans-unit id="6">
  <source>and %count% more...</source>
  <target>et %count% autres...</target>
</trans-unit>
```

La seule exigence dans la chaîne traduite est d'utiliser l'espace réservé %count% quelque part.

Certaines autres chaînes sont encore plus complexes car ils impliquent les pluriels. Selon certains chiffres, la syntaxe change, mais pas nécessairement de la même façon pour toutes les langues. Certaines langues ont des règles de grammaire très complexe pour les pluriels, comme le polonais ou le russe.

Sur la page de catégorie, le nombre d'emplois dans la catégorie actuelle est affichée :

```
<!-- apps/frontend/modules/category/templates/showSuccess.php -->
<strong><?php echo count($pager) ?></strong> jobs in this category
```

Lorsqu'une phrase a différentes traductions, en fonction du nombre, le helper `format_number_choice()` doit être utilisée :

```
<?php echo format_number_choice(
    '[0]No job in this category|[1]One job in this category|(1,+Inf]%count% jobs in
this category',
```

```
    array('%count%' => '<strong>'.count($pager).'</strong>'),
    count($pager)
  )
?>
```

Le Helper `format_number_choice()` prend trois arguments :

- La chaîne à utiliser en fonction du nombre
- Un tableau d'espace réservé
- Le numéro à utiliser pour déterminer le texte à utiliser

La chaîne qui décrit les différentes traductions en fonction du nombre est formaté comme suit :

- Chaque possibilité est séparée par un caractère pipe (`|`)
- Chaque chaîne est composée d'une portée suivie de la traduction

La portée peut décrire n'importe quel éventail de nombres :

- `[1,2]`: Accepte les valeurs entre 1 et 2 inclus
- `(1,2)`: Accepte des valeurs comprises entre 1 et 2 en excluant 1 et 2
- `{1,2,3,4}`: Seules les valeurs définies sont acceptées
- `[-Inf,0)`: Accepte les valeurs supérieures ou égales à l'infini négatif et strictement inférieur à 0
- `{n: n % 10 > 1 && n % 10 < 5}`: correspond à des nombres comme 2, 3, 4, 22, 23, 24

La traduction de la chaîne est similaire à d'autres chaînes de message :

```
<trans-unit id="7">
  <source>[0]No job in this category|[1]One job in this category|(1,+Inf]%count% jobs
in this category</source>
  <target>[0]Aucune annonce dans cette catégorie|[1]Une annonce dans cette
catégorie|(1,+Inf]%count% annonces dans cette catégorie</target>
</trans-unit>
```

Maintenant que vous savez comment internationaliser toutes sortes de chaines, prenez le temps d'ajouter l'appel `__()` à tous les Templates de l'application frontend. Nous n'allons pas internationaliser l'application backend.

Formulaires

Les classes de formulaires contiennent de nombreuses chaines qui doivent être traduites, comme les labels, les messages d'erreur et les messages d'aide. Toutes ces chaînes sont automatiquement internationalisées par symfony, donc vous avez seulement besoin de fournir des traductions dans les fichiers XLIFF.

Malheureusement, la tâche `i18n:extract` ne sait pas encore analyser les classes de formulaires pour les chaînes non traduites.

Les objets de Doctrine

Pour le site web Jobeet, nous n'allons pas internationaliser tous les tables car il n'y a pas de sens de demander aux annonceurs de traduire leurs annonces d'emploi dans toutes les langues disponibles. Mais la table des catégories doit absolument être traduit.

Le plugin de Doctrine supporte la sortie des tables i18n. Pour chaque table qui contient des données localisées, deux tables doivent être créés : une pour les colonnes qui sont indépendantes de l'i18n, et l'autre avec les colonnes qui doivent être internationalisé. Les deux tables sont reliées par une relation 1-n.

Mettez à jour le `schema.yml|schema.yml (I18n)` en conséquence :

```
# config/doctrine/schema.yml
JobeetCategory:
  actAs:
    Timestampable: ~
    I18n:
      fields: [name]
      actAs:
        Sluggable: { fields: [name], uniqueBy: [lang, name] }
  columns:
    name: { type: string(255), notnull: true }
```

En se tournant sur le comportement I18n, un modèle nommé JobeetCategoryTranslation sera automatiquement créé et les fields spécifiés sont déplacés vers ce modèle.

Remarquez que nous mettons simplement le comportement I18n et déplacons le comportement Sluggable pour être attaché au modèle JobeetCategoryTranslation automatiquement créé. L'option uniqueBy décrit pour le comportement Sluggable quels sont les champs qui déterminent si un slug est unique ou non. Dans notre cas, chaque slug doit être unique pour chaque paire lang et name.

Et actualisez les fixtures pour les catégories:

```
# data/fixtures/categories.yml
JobeetCategory:
  design:
    Translation:
      en:
        name: Design
      fr:
        name: design
  programming:
    Translation:
      en:
        name: Programming
      fr:
        name: Programmation
  manager:
    Translation:
      en:
        name: Manager
      fr:
        name: Manager
  administrator:
    Translation:
      en:
        name: Administrator
      fr:
        name: Administrateur
```

Nous avons besoin aussi de surcharger la méthode findOneBySlug() dans JobeetCategoryTable. Car Doctrine fournit quelques chercheurs magiques pour toutes les colonnes dans un modèle. Nous avons besoin de créer simplement la méthode findOneBySlug() afin de surcharger la fonctionnalité magique par défaut qu'offre Doctrine.

Nous devons faire quelques modifications afin que la catégorie soit récupérée sur la base du slug anglais dans la table JobeetCategoryTranslation.

```
// lib/model/doctrine/JobeetCategoryTable.cass.php
public function findOneBySlug($slug)
{
  $q = $this->createQuery('a')
    ->leftJoin('a.Translation t')
    ->andWhere('t.lang = ?', 'en')
    ->andWhere('t.slug = ?', $slug);
  return $q->fetchOne();
}
```

Reconstruisez le modèle :

```
$ php symfony doctrine:build --all --and-load --no-confirmation
$ php symfony cc
```

Comme la tâche doctrine:build --all --and-load supprime toutes les tables et les données de la base de données, n'oubliez pas de re-créer un utilisateur pour accéder au backend de Jobeet avec la tâche guard:create-user. Autrement, vous pouvez ajouter un fichier fixture pour l'ajouter automatiquement pour vous.

Lorsque vous utilisez le comportement I18n, les proxies sont créés entre l'objet JobeetCategory et l'objet JobeetCategoryTranslation, donc toutes les anciennes fonctions pour récupérer le nom de la catégorie fonctionneront encore et récupéront la valeur pour la culture actuelle.

```
$category = new JobeetCategory();
```

```
$category->setName('foo'); // sets the name for the current culture
$category->getName(); // gets the name for the current culture

$this->getUser()->setCulture('fr'); // from your actions class

$category->setName('foo'); // sets the name for French
echo $category->getName(); // gets the name for French
```

Pour réduire le nombre de requêtes à la base de données, joignez JobeetCategoryTranslation dans vos requêtes. Cela permettra de récupérer l'objet principal et celui du i18n dans une seule requête.

```
$categories = Doctrine_Query::create()
  ->from('JobeetCategory c')
  ->leftJoin('c.Translation t WITH t.lang = ?', $culture)
  ->execute();
```

Le mot clé WITH ci-dessus va ajouter une condition à la condition ON ajoutée automatiquement sur de la requête. Ainsi, la condition ON de la jointure sera à la fin.

```
LEFT JOIN c.Translation t ON c.id = t.id AND t.lang = ?
```

Comme la route category est liée à la classe du modèle JobeetCategory et parce que le slug fait maintenant partie de JobeetCategoryTranslation, la route n'est pas en mesure de récupérer l'objet Category automatiquement. Pour aider le système de routage, nous allons créer une méthode qui se chargera de la récupération de l'objet :

Comme nous avions déjà surchargé findOneBySlug(), refactorisons un peu plus afin que ces méthodes puissent être partagées. Nous allons créer des nouvelles méthodes findOneBySlugAndCulture() et doSelectForSlug() et changer la méthode findOneBySlug() pour utiliser simplement la méthode findOneBySlugAndCulture().

```
// lib/model/doctrine/JobeetCategoryTable.class.php
public function doSelectForSlug($parameters)
{
  return $this->findOneBySlugAndCulture($parameters['slug'],
$parameters['sf_culture']);
}

public function findOneBySlugAndCulture($slug, $culture = 'en')
{
  $q = $this->createQuery('a')
    ->leftJoin('a.Translation t')
    ->andWhere('t.lang = ?', $culture)
    ->andWhere('t.slug = ?', $slug);
  return $q->fetchOne();
}

public function findOneBySlug($slug)
{
  return $this->findOneBySlugAndCulture($slug, 'en');
}
```

Ensuite, utilisez l'option methodmethod (Routage) pour dire à la route category d'utiliser la méthode doSelectForSlug() pour récupérer l'objet :

```
# apps/frontend/config/routing.yml
category:
  url:     /:sf_culture/category/:slug.:sf_format
  class:   sfDoctrineRoute
  param:   { module: category, action: show, sf_format: html }
  options: { model: JobeetCategory, type: object, method: doSelectForSlug }
  requirements:
    sf_format: (?:html|atom)
```

Nous avons besoin de recharger les jeux de test pour régénérer les slugs adéquates pour les catégories :

```
$ php symfony doctrine:data-load
```

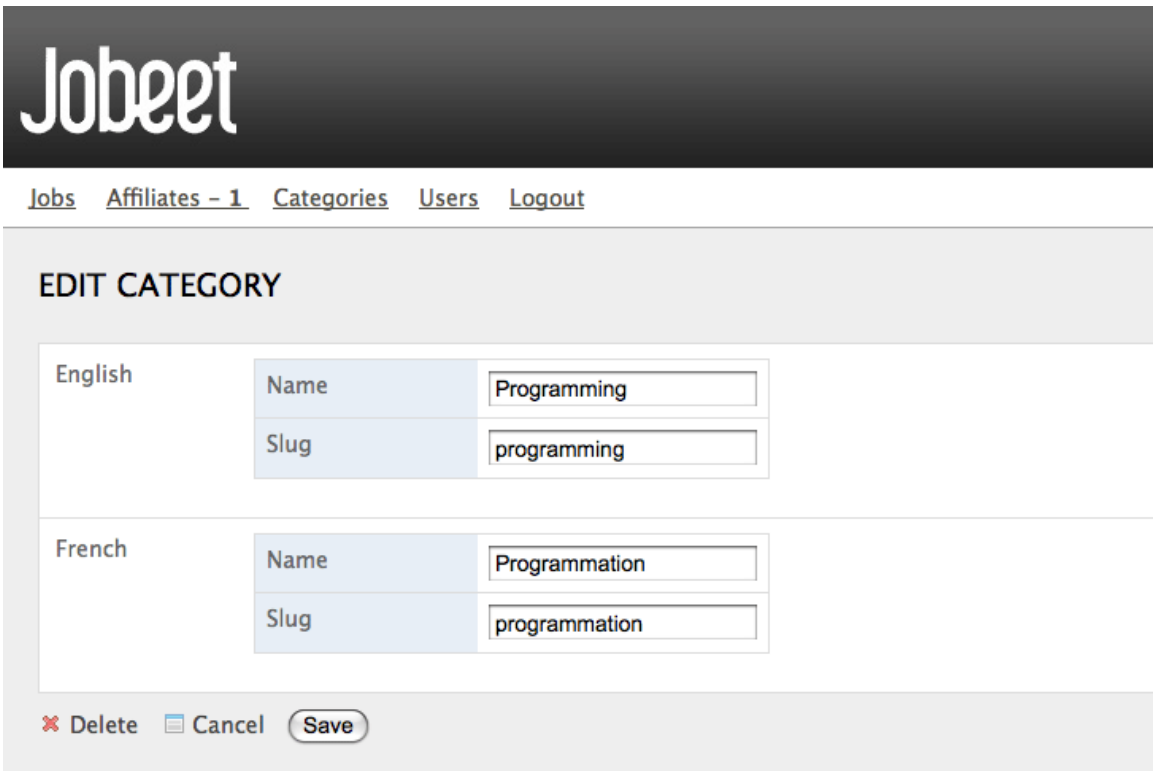
Maintenant, la route category est internationalisé et l'URL pour une catégorie intègre le slug de

la catégorie traduite :

```
/frontend_dev.php/fr/category/programmation
/frontend_dev.php/en/category/programming
```

Admin Generator

Pour le backend, nous voulons que les traductions françaises et anglaises soient éditées dans le même formulaire :



L'intégration d'un formulaire i18n peut être fait en utilisant la méthode `embedI18N()` :

```
// lib/form/JobeetCategoryForm.class.php
class JobeetCategoryForm extends BaseJobeetCategoryForm
{
  public function configure()
  {
    unset(
      $this['jobeet_affiliates_list'],
      $this['created_at'], $this['updated_at']
    );

    $this->embedI18n(array('en', 'fr'));
    $this->widgetSchema->setLabel('en', 'English');
    $this->widgetSchema->setLabel('fr', 'French');
  }
}
```

L'interface de l'admin generator supporte l'internationalisation. Il est livré avec des traductions pour plus de 20 langues, et il est très facile d'en ajouter une nouvelle, ou pour d'en personnaliser une existante. Copiez le fichier pour la langue que vous souhaitez personnaliser depuis symfony (les traductions de l'admin se trouvent dans `lib/vendor/symfony/lib/plugins/sfDoctrinePlugin/i18n/`) dans le répertoire `i18n` de l'application. Comme le fichier dans votre application sera fusionné avec celle de symfony, ne conservez que les chaines modifiées dans le fichier de l'application.

Vous remarquerez que les fichiers de traduction de l'admin generator sont nommés `sf_admin.fr.xml`, au lieu de `fr/messages.xml`. En réalité, `messages` est le nom du catalogue par défaut utilisé par symfony et il peut être modifié pour permettre une meilleure séparation entre les différentes parties de votre application. En utilisant un autre catalogue que celui par défaut, cela nécessite que vous le spécifier lorsque vous utilisez le helper `__()` :

```
<?php echo __('About Jobeet', array(), 'jobeet') ?>
```

Dans l'appel précédent de `__()`, symfony va chercher la chaîne "About Jobeet" dans le catalogue `jobeet`.

Tests

La correction des tests est une partie intégrante de la migration de l'internationalisation. Premièrement, mettez à jour les jeux de test pour les catégories en copiant les jeux de test que nous avons défini ci-dessus dans `test/fixtures/categories.yml`.

N'oubliez pas de mettre à jour les méthodes dans le fichier `lib/test/JobeetTestFunctional.class.php` afin de prendre en compte les modifications apportées lors de l'internationalisation du modèle `JobeetCategory`.

```
public function getMostRecentProgrammingJob()
{
  $q = Doctrine_Query::create()
    ->select('j.*')
    ->from('JobeetJob j')
    ->leftJoin('j.JobeetCategory c')
    ->leftJoin('c.Translation t')
    ->where('t.slug = ?', 'programming');

  $q = Doctrine_Core::getTable('JobeetJob')->addActiveJobsQuery($q);

  return $q->fetchOne();
}
```

Reconstruisez le modèle de l'environnement de `test` :

```
$ php symfony doctrine:build --all --no-confirmation --env=test
```

Vous pouvez maintenant lancer tous les tests pour vérifier qu'ils s'exécutent bien :

```
$ php symfony test:all
```

Quand nous avons développé l'interface backend pour Jobeet, nous n'avons pas écrit les tests fonctionnels. Mais chaque fois que vous créez un module avec la ligne de commande de symfony, symfony produit aussi des bouts de test. Ces bouts sont sûres d'être enlevés.

Régionalisation

Templates

Le support des différentes cultures, c'est aussi le soutien de différents formats pour les dates et les chiffres. Dans un Template, plusieurs helpers sont à votre disposition pour vous aider à prendre en compte toutes ces différences, basée sur la culture actuelle de l'utilisateur :

Dans le groupe d'helper `Date` :

Helper	Description
`format_date()`	Formate la date
`format_datetime()`	Formate la date avec l'heure (heures, minutes, secondes)
`time_ago_in_words()`	Affiche le temps écoulé entre une date et maintenant
`distance_of_time_in_words()`	Affiche le temps écoulé entre deux dates
`format_daterange()`	Formate un intervalle de dates

Dans le groupe d'helper `Number` :

Helper	Description
`format_number()`	Formate un nombre
`format_currency()`	Formate une monnaie

Dans le groupe d'helper `I18N` :

Helper	Description
`format_country()`	Affiche le nom d'un pays
`format_language()`	Affiche le nom d'une langue

Formulaires (I18n)

Le framework de formulaire fournit plusieurs widgets et validateurs pour régionalisé les données :

- sfWidgetFormI18nDate
- sfWidgetFormI18nDateTime
- sfWidgetFormI18nTime
- sfWidgetFormI18nChoiceCountry
- sfWidgetFormI18nChoiceCurrency
- sfWidgetFormI18nChoiceLanguage
- sfWidgetFormI18nChoiceTimezone
- sfValidatorI18nChoiceCountry
- sfValidatorI18nChoiceLanguage
- sfValidatorI18nChoiceTimezone

À demain

L'internationalisation et la régionalisation sont des citoyens de première classe dans symfony. Fournir un site régionalisé à vos utilisateurs est très facile car symfony fournit tous les outils de base et vous donne même les tâches en ligne de commande pour le faire rapidement.

Soyez prêts pour un tutoriel très spécial demain où nous déplacerons beaucoup de fichiers et explorons une approche différente de l'organisation d'un projet symfony.

« Jour 18 : AJAX

Jour 20 : Les Plugins »

Questions & Feedback

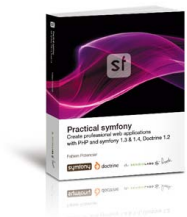
If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 20 : Les Plugins

Hier, vous avez appris à internationaliser et régionaliser vos applications symfony. Encore une fois, grâce à la norme ICU et beaucoup de helpers, symfony le fait vraiment facilement.

Aujourd'hui, nous allons parler des plugins : ce qu'ils sont, ce que vous pouvez empaqueter dans un plugin et comment ils peuvent être utilisés.



Plugins

Un plugin symfony

Un plugin symfony offre une façon d'empaqueter et distribuer une partie de vos fichiers du projet. Comme un projet, un plugin peut contenir des classes, des helpers, de la configuration, des tâches, des modules, des schémas et même des ressources web.

Plugins privés

La première utilisation des plugins est de faciliter le partage de code entre vos applications, ou même entre différents projets. Rappelons que les applications symfony ne partagent que le modèle. Les plugins fournissent un moyen de partager plus de composants entre les applications.

Si vous avez besoin de réutiliser le même schéma pour des projets différents, ou les mêmes modules, déplacez les dans un plugin. Comme un plugin est simplement un répertoire, vous pouvez le déplacer très facilement en créant un dépôt SVN et en utilisant `svn:externals`, ou en copiant simplement les fichiers d'un projet à l'autre.

Nous les appelons "plugins privés" parce que leur utilisation est restreinte à un unique développeur ou entreprise. Ils ne sont pas accessibles au public.

Vous pouvez même créer un package de vos plugins privé, créer votre propre canal de plugin symfony et les installer via la tâche `plugin:install`.

Plugins publiques

Les plugins publiques sont disponibles pour la communauté pour être télécharger et installer. Au cours de ce tutoriel, nous avons utilisé un couple de plugins publiques : `sfDoctrineGuardPlugin` et `sfFormExtraPlugin`.

Ce sont exactement les mêmes que pour les plugins privés. La seule différence est que n'importe qui peut les installer dans leurs projets. Vous apprendrez plus tard la manière de publier et d'héberger un plugin publique sur le site de symfony.

Une façon différente d'organiser le code

Il y a une autre façon de voir les plugins et comment les utiliser. Oubliez la réutilisation et le partage. Les plugins peuvent être utilisées d'une façon différente pour organiser votre code. Au lieu d'organiser les fichiers par couche : tous les modèles dans le répertoire `lib/model/`, les templates dans le répertoire `templates/`, ...; les fichiers sont mis en place par fonction : tous les fichiers emplois (le modèle, les modules et les templates), tous les fichiers CMS, et ainsi de suite.

Structure des fichiers du plugin

Un plugin est juste une structure de répertoire avec des fichiers organisés selon une structure prédéfinie, selon la nature des fichiers. Aujourd'hui, nous allons passer la plupart du code que nous avons écrit pour Jobeet dans un `sfJobeetPlugin`. La structure de base que nous allons utiliser est la suivante :

```
sfJobeetPlugin/
```

```
  config/
    sfJobeetPluginConfiguration.class.php // Plugin initialization
    routing.yml                           // Routing
    doctrine/
      schema.yml                          // Database schema
  lib/
    Jobeet.class.php                      // Classes
    helper/                               // Helpers
    filter/                               // Filter classes
    form/                                 // Form classes
    model/                                // Model classes
    task/                                 // Tasks
  modules/
    job/                                  // Modules
      actions/
      config/
      templates/
  web/                                    // Assets like JS, CSS, and images
```

Le plugin Jobeet

L'initialisation d'un plugin est aussi simple que de créer un nouveau répertoire sous `plugins/`. Pour Jobeet, nous allons créer un répertoire `sfJobeetPlugin` :

```
$ mkdir plugins/sfJobeetPlugin
```

Ensuite, activez le plugin `sfJobeetPlugin` dans le fichier `config/ProjectConfiguration.class.php`.

```
public function setup()
{
  $this->enablePlugins(array(
    'sfDoctrinePlugin',
    'sfDoctrineGuardPlugin',
    'sfFormExtraPlugin',
    'sfJobeetPlugin'
  ));
}
```

Tous les plugins doivent se terminer par un `Plugin`. C'est aussi une bonne habitude de les préfixer avec `sf`, même si elle n'est pas obligatoire.

Le Modèle

Premièrement, déplacez le fichier `config/doctrine/schema.yml` vers `plugins/sfJobeetPlugin/config/` :

```
$ mkdir plugins/sfJobeetPlugin/config/
$ mkdir plugins/sfJobeetPlugin/config/doctrine
$ mv config/doctrine/schema.yml plugins/sfJobeetPlugin/config/doctrine/schema.yml
```

Toutes les commandes sont pour Unix et assimilés. Si vous utilisez Windows, vous pouvez glisser et déposer des fichiers dans l'explorateur. Et si vous utilisez Subversion, ou tout autre outil pour gérer votre code, utilisez les outils intégrés qu'ils fournissent (comme `svn mv` pour déplacer des fichiers).

Déplacez les fichiers du modèle, des formulaires et des filtres vers `plugins/sfJobeetPlugin/lib/`:

```
$ mkdir plugins/sfJobeetPlugin/lib/
$ mv lib/model/ plugins/sfJobeetPlugin/lib/
$ mv lib/form/ plugins/sfJobeetPlugin/lib/
$ mv lib/filter/ plugins/sfJobeetPlugin/lib/


$ rm -rf plugins/sfJobeetPlugin/lib/model/doctrine/sfDoctrineGuardPlugin
$ rm -rf plugins/sfJobeetPlugin/lib/form/doctrine/sfDoctrineGuardPlugin
$ rm -rf plugins/sfJobeetPlugin/lib/filter/doctrine/sfDoctrineGuardPlugin
```

Supprimez le fichier `plugins/sfJobeetPlugin/lib/form/BaseForm.class.php`.

```
$ rm plugins/sfJobeetPlugin/lib/form/BaseForm.class.php
```

Après avoir déplacé les modèles, les formulaires et les filtre, les classes doivent être renommées, passez les en classes abstraites et préfixez les par le mot `Plugin`.

Préfixez seulement les classes auto-générées avec `Plugin` et non pas toutes les classes. Par exemple ne faites pas précéder toutes les classes que vous avez écrit à la main. Seules les classes auto-générés nécessitent le préfixe.

Voici un exemple où il faut déplacer les classes `JobeetAffiliate` et `JobeetAffiliateTable`.

```
$ mv plugins/sfJobeetPlugin/lib/model/doctrine/JobeetAffiliate.class.php
plugins/sfJobeetPlugin/lib/model/doctrine/PluginJobeetAffiliate.class.php
```

Et le code doit être mis à jour :

```
abstract class PluginJobeetAffiliate extends BaseJobeetAffiliate
{
  public function save(Doctrine_Connection $conn = null)
  {
    if (!$this->getToken())
    {
      $this->setToken(sha1($object->getEmail().rand(11111, 99999)));
    }

    parent::save($conn);
  }

  // ...
}
```

Maintenant déplaçons la classe `JobeetAffiliateTable` :

```
$ mv plugins/sfJobeetPlugin/lib/model/doctrine/JobeetAffiliateTable.class.php
plugins/sfJobeetPlugin/lib/model/doctrine/PluginJobeetAffiliateTable.class.php
```

La définition de la classe devrait ressembler à ce qui suit :

```
abstract class PluginJobeetAffiliateTable extends Doctrine_Table
{
  // ...
}
```

Maintenant, faites la même chose pour les classes de formulaire et de filtre. Renommez-les pour inclure un préfixe avec le mot `Plugin`.

Assurez-vous de supprimer le répertoire `base` dans `plugins/sfJobeetPlugin/lib/*/doctrine/` pour les répertoires `form`, `filter` et `model` :

```
$ rm -rf plugins/sfJobeetPlugin/lib/form/doctrine/base
$ rm -rf plugins/sfJobeetPlugin/lib/filter/doctrine/base
$ rm -rf plugins/sfJobeetPlugin/lib/model/doctrine/base
```

Une fois que vous avez déplacé, renommé et supprimé certains formulaires, filtres et classes du modèle, exécutez les tâches pour re-construire toutes les classes :

```
$ php symfony doctrine:build --all-classes
```

Maintenant, vous remarquerez quelques nouveaux répertoires créés qui détiennent des modèles créés à partir du schéma fourni avec le `sfJobeetPlugin` dans `lib/model/doctrine/sfJobeetPlugin/`.

Ce répertoire contient les modèles de haut niveau et les classes de base générées à partir du schéma. Par exemple le modèle `JobeetJob` a maintenant cette structure de classe :

- `JobeetJob` (étend `PluginJobeetJob`) dans `lib/model/doctrine/sfJobeetPlugin/JobeetJob.class.php`: La classe de haut niveau où toutes les fonctionnalités du modèle du projet peuvent être placées. C'est là que vous pouvez ajouter ou remplacer une fonctionnalité qui est fournie avec le modèle du plugin.
- `PluginJobeetJob` (étend `BaseJobeetJob`) dans

`plugins/sfJobeetPlugin/lib/model/doctrine/PluginJobeetJob.class.php`: Cette classe contient toutes les fonctionnalités spécifiques du plugin. Vous pouvez surcharger la fonctionnalité dans cette classe et la base en modifiant la classe `JobeetJob`.

- `BaseJobeetJob` (étend `sfDoctrineRecord`) dans `lib/model/doctrine/sfJobeetPlugin/base/BaseJobeetJob.class.php`: La classe de base qui est générée depuis le fichier yaml du schéma chaque fois que vous exécutez `doctrine:build --model`.
- `JobeetJobTable` (étend `PluginJobeetJobTable`) dans `lib/model/doctrine/sfJobeetPlugin/JobeetJobTable.class.php`: Identique à la classe `JobeetJob` sauf que c'est l'instance de `Doctrine_Table` qui sera retourné lorsque vous appelez `Doctrine_Core::getTable('JobeetJob')`.
- `PluginJobeetJobTable` (étend `Doctrine_Table`) dans `lib/model/doctrine/sfJobeetPlugin/JobeetJobTable.class.php`: Cette classe contient toutes les fonctionnalités spécifiques du plugin pour l'instance de `Doctrine_Table` qui sera retourné lorsque vous appelez `Doctrine_Core::getTable('JobeetJob')`.

Avec cette structure générée, vous avez la possibilité de personnaliser les modèles d'un plugin en éditant le haut niveau de la classe `JobeetJob`. Vous pouvez personnaliser le schéma et ajouter des colonnes, ajouter des relations en surchargeant les méthodes `setTableDefinition()` et `setUp()`.

Lorsque vous déplacez les classes de formulaires, n'oubliez pas de changer la méthode `configure()` par la méthode `setup()` et appelez `parent::setup()`. Ci-dessous, un exemple.

```
abstract class PluginJobeetAffiliateForm extends BaseJobeetAffiliateForm
{
  public function setup()
  {
    parent::setup();
  }

  // ...
}
```

Nous devons nous assurer que notre plugin n'a pas les classes de base pour tous les formulaires de Doctrine. Ces fichiers sont globaux pour un projet et seront re-générés avec `doctrine:build --forms` et `doctrine:build --filters`.

Supprimer les fichiers du plugin :

```
$ rm plugins/sfJobeetPlugin/lib/form/doctrine/BaseFormDoctrine.class.php
$ rm plugins/sfJobeetPlugin/lib/filter/doctrine/BaseFormFilterDoctrine.class.php
```

Vous pouvez également déplacer le fichier `Jobeet.class.php` vers le plugin :

```
$ mv lib/Jobeet.class.php plugins/sfJobeetPlugin/lib/
```

Comme nous avons déplacé des fichiers, videz le cache :

```
$ php symfony cc
```

Si vous utilisez un accélérateur PHP comme APC, les choses deviennent étranges à ce stade, redémarrez Apache.

Maintenant que tous les fichiers du modèle ont été déplacés dans le plugin, exécuter les tests pour vérifier que tout fonctionne toujours très bien :

```
$ php symfony test:all
```

Les contrôleurs et les vues

L'étape logique suivante consiste à déplacer les modules vers le plugin. Pour éviter les collisions de nom de module, c'est une bonne habitude de faire précéder les noms des modules du plugin par le nom du plugin :

```
$ mkdir plugins/sfJobeetPlugin/modules/
$ mv apps/frontend/modules/affiliate plugins/sfJobeetPlugin/modules/sfJobeetAffiliate
```

```
$ mv apps/frontend/modules/api plugins/sfJobeetPlugin/modules/sfJobeetApi
$ mv apps/frontend/modules/category plugins/sfJobeetPlugin/modules/sfJobeetCategory
$ mv apps/frontend/modules/job plugins/sfJobeetPlugin/modules/sfJobeetJob
$ mv apps/frontend/modules/language plugins/sfJobeetPlugin/modules/sfJobeetLanguage
```

Pour chaque module, vous devez également modifier le nom des classes dans toutes les fichiers `actions.class.php` et `components.class.php` (par exemple, la classe `affiliateActions` doit être renommé en `sfJobeetAffiliateActions`).

Les appels `include_partial()` et `include_component()` doivent aussi être changés dans les Templates suivants :

- `sfJobeetAffiliate/templates/_form.php` (changez `affiliate` en `sfJobeetAffiliate`)
- `sfJobeetCategory/templates/showSuccess.atom.php`
- `sfJobeetCategory/templates/showSuccess.php`
- `sfJobeetJob/templates/indexSuccess.atom.php`
- `sfJobeetJob/templates/indexSuccess.php`
- `sfJobeetJob/templates/searchSuccess.php`
- `sfJobeetJob/templates/showSuccess.php`
- `apps/frontend/templates/layout.php`

Mettez à jour les actions search et delete :

```
// plugins/sfJobeetPlugin/modules/sfJobeetJob/actions/actions.class.php
class sfJobeetJobActions extends sfActions
{
  public function executeSearch(sfWebRequest $request)
  {
    $this->forwardUnless($query = $request->getParameter('query'), 'sfJobeetJob',
'index');

    $this->jobs = Doctrine_Core::getTable('JobeetJob') ->getForLuceneQuery($query);

    if ($request->isXmlHttpRequest())
    {
      if ('*' == $query || !$this->jobs)
      {
        return $this->renderText('No results.');
      }

      return $this->renderPartial('sfJobeetJob/list', array('jobs' => $this->jobs));
    }
  }

  public function executeDelete(sfWebRequest $request)
  {
    $request->checkCSRFProtection();

    $jobeet_job = $this->getRoute()->getObject();
    $jobeet_job->delete();

    $this->redirect('sfJobeetJob/index');
  }

  // ...
}
```

Maintenant, modifiez le fichier `routing.yml` pour prendre ces changements en compte :

```
# apps/frontend/config/routing.yml
affiliate:
  class:   sfDoctrineRouteCollection
  options:
    model:           JobeetAffiliate
    actions:         [new, create]
    object_actions: { wait: GET }
    prefix_path:     /:sf_culture/affiliate
    module:          sfJobeetAffiliate
  requirements:
    sf_culture: (?:fr|en)
```

```
api_jobs:
  url:     /api/:token/jobs.:sf_format
  class:   sfDoctrineRoute
  param:   { module: sfJobeetApi, action: list }
  options: { model: JobeetJob, type: list, method: getForToken }
  requirements:
    sf_format: (?:xml|json|yaml)

category:
  url:     /:sf_culture/category/:slug.:sf_format
  class:   sfDoctrineRoute
  param:   { module: sfJobeetCategory, action: show, sf_format: html }
  options: { model: JobeetCategory, type: object, method: doSelectForSlug }
  requirements:
    sf_format: (?:html|atom)
    sf_culture: (?:fr|en)

job_search:
  url:   /:sf_culture/search
  param: { module: sfJobeetJob, action: search }
  requirements:
    sf_culture: (?:fr|en)

job:
  class:   sfDoctrineRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: PUT, extend: PUT }
    prefix_path:    /:sf_culture/job
    module:         sfJobeetJob
  requirements:
    token: \w+
    sf_culture: (?:fr|en)

job_show_user:
  url:     /:sf_culture/job/:company_slug/:location_slug/:id/:position_slug
  class:   sfDoctrineRoute
  options:
    model: JobeetJob
    type: object
    method_for_query: retrieveActiveJob
  param:   { module: sfJobeetJob, action: show }
  requirements:
    id:        \d+
    sf_method: GET
    sf_culture: (?:fr|en)

change_language:
  url:   /change_language
  param: { module: sfJobeetLanguage, action: changeLanguage }

localized_homepage:
  url:   /:sf_culture/
  param: { module: sfJobeetJob, action: index }
  requirements:
    sf_culture: (?:fr|en)

homepage:
  url:   /
  param: { module: sfJobeetJob, action: index }
```

Si vous essayez de parcourir le site Jobeet maintenant, vous allez avoir des exceptions vous indiquant que les modules ne sont pas activés. Comme les plugins sont partagés par toutes les applications dans un projet, vous devez activer spécifiquement le module dont vous avez besoin pour une application donnée dans le fichier de configuration `settings.yml` :

```
# apps/frontend/config/settings.yml
all:
  .settings:
    enabled_modules:
      - default
```

```
        - sfJobeetAffiliate
        - sfJobeetApi
        - sfJobeetCategory
        - sfJobeetJob
        - sfJobeetLanguage
```

La dernière étape de la migration est de fixer les tests fonctionnels où nous les testons pour le nom du module.

Les tâches

Les tâches peuvent être déplacés vers le plugin assez facilement :

```
$ mv lib/task plugins/sfJobeetPlugin/lib/
```

Les fichiers i18n

Un plugin peut aussi contenir des fichiers XLIFF :

```
$ mv apps/frontend/i18n plugins/sfJobeetPlugin/
```

Le Routage

Un plugin peut également contenir des règles de routage :

```
$ mv apps/frontend/config/routing.yml plugins/sfJobeetPlugin/config/
```

Les ressources

Même si c'est un peu contre-intuitif, un plugin peut également contenir des ressources web comme les images, les feuilles de style et les JavaScripts. Comme nous ne voulons pas distribuer le plugin Jobeet, cela n'a pas vraiment de sens, mais cela est possible en créant un répertoire `plugins/sfJobeetPlugin/web/`.

Les ressources d'un plugin doit être accessible dans un répertoire web/ pour être lisible à partir d'un navigateur. Le `plugin:publish-assets` adresse cela en créant des liens symboliques sous le système Unix et en copiant les fichiers sur la plate-forme Windows :

```
$ php symfony plugin:publish-assets
```

L'utilisateur

Le déplacement des méthodes de la classe `myUser` qui traitent de l'historique des emplois est un peu plus compliqué. Nous pourrions créer une classe `JobeetUser` et faire hérité `myUser`. Mais il y a un meilleur moyen, surtout si plusieurs plugins souhaitent ajouter de nouvelles méthodes pour la classe.

Les objets du noyau de symfony notifient les événements au cours de leur cycle de vie que vous pouvez écouter. Dans notre cas, nous devons écouter l'événement `user.method_not_found`, qui se produit lorsqu'une méthode non définie est appelée sur l'objet `sfUser`.

Quand symfony est initialisé, tous les plugins sont également initialisés s'ils ont une classe de configuration de plugin :

```
// plugins/sfJobeetPlugin/config/sfJobeetPluginConfiguration.class.php
class sfJobeetPluginConfiguration extends sfPluginConfiguration
{
  public function initialize()
  {
    $this->dispatcher->connect('user.method_not_found', array('JobeetUser',
'methodNotFound'));
  }
}
```

Les notifications d'événements sont gérés par `sfEventDispatcher`, l'objet du dispatcher d'événement. L'enregistrement d'un listener est aussi simple que d'appeler la méthode `connect()`. La méthode `connect()` connecte un nom d'événement à un PHP appelable.

Un PHP appelable est une variable PHP qui peut être utilisée par la fonction `call_user_func()` et renvoie `true` lorsque elle est passée à la fonction `is_callable()`. Une chaîne représente une fonction et un tableau peut représenter une méthode d'objet ou une méthode de classe.

Avec le code en place ci-dessus, l'objet `myUser` appelera la méthode statique `methodNotFound()` de la classe `JobeetUser` chaque fois qu'il est incapable de trouver une méthode. Il appartient ensuite à la méthode `methodNotFound()` de traiter la méthode manquante ou non.

Supprimez toutes les méthodes de la classe `myUser` et créez la classe `JobeetUser` :

```
// apps/frontend/lib/myUser.class.php
class myUser extends sfBasicSecurityUser
{
}

// plugins/sfJobeetPlugin/lib/JobeetUser.class.php
class JobeetUser
{
  static public function methodNotFound(sfEvent $event)
  {
    if (method_exists('JobeetUser', $event['method']))
    {
      $event->setReturnValue(call_user_func_array(
        array('JobeetUser', $event['method']),
        array_merge(array($event->getSubject()), $event['arguments'])
      ));

      return true;
    }
  }

  static public function isFirstRequest(sfUser $user, $boolean = null)
  {
    if (is_null($boolean))
    {
      return $user->getAttribute('first_request', true);
    }
    else
    {
      $user->setAttribute('first_request', $boolean);
    }
  }

  static public function addJobToHistory(sfUser $user, JobeetJob $job)
  {
    $ids = $user->getAttribute('job_history', array());

    if (!in_array($job->getId(), $ids))
    {
      array_unshift($ids, $job->getId());
      $user->setAttribute('job_history', array_slice($ids, 0, 3));
    }
  }

  static public function getJobHistory(sfUser $user)
  {
    $ids = $user->getAttribute('job_history', array());

    if (!empty($ids))
    {
      return Doctrine_Core::getTable('JobeetJob')
        ->createQuery('a')
        ->whereIn('a.id', $ids)
        ->execute();
    }

    return array();
  }

  static public function resetJobHistory(sfUser $user)
  {
    $user->getAttributeHolder()->remove('job_history');
  }
}
```

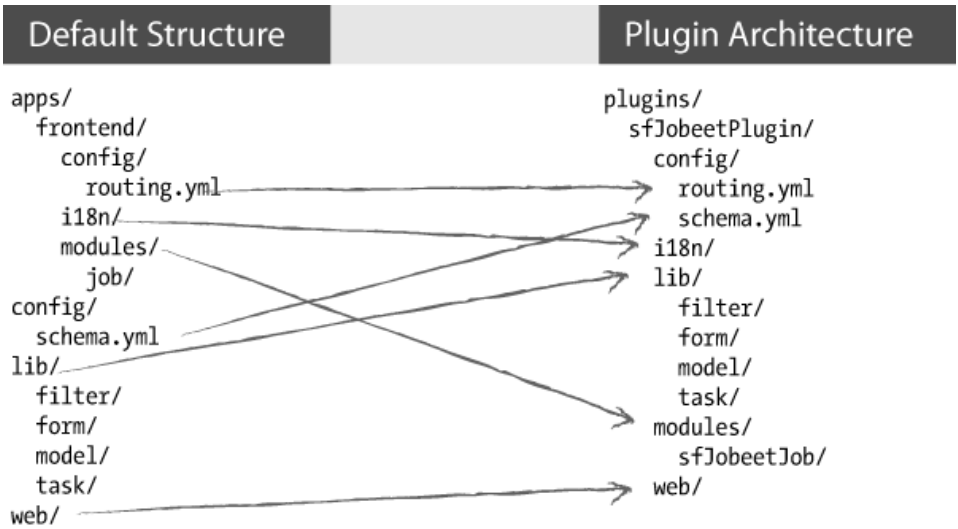
Lorsque le dispatcher appelle la méthode `methodNotFound()`, il passe un objet `sfEvent`.

Si la méthode existe dans la classe `JobeetUser`, il est appelé et sa valeur retournée est retournée par la suite au notificateur. Sinon, symfony va essayer le listener suivant enregistré ou lever une exception.

La méthode `getSubject()` retourne le notificateur de l'événement, qui dans ce cas est l'objet actuel `myUser`.

La structure par défaut contre l'architecture du plugin

L'utilisation de l'architecture du plugin vous permet d'organiser votre code d'une manière différente :



L'utilisation du plugin

Lorsque vous commencez à mettre en œuvre une nouvelle fonctionnalité, ou si vous essayez de résoudre un problème web classique, y a fort à parier que quelqu'un a déjà résolu le même problème et peut-être emballé la solution dans un plugin symfony. Pour rechercher un plugin publique de symfony, accédez à la section des plugins du site de symfony.

Comme un plugin est contenu dans un répertoire, il y a plusieurs façons de l'installer :

- Utiliser la tâche `plugin:install` (cela ne fonctionne que si le développeur du plugin a créé un package du plugin et l'a envoyé sur le site symfony)
- Le téléchargement du package et la décompression manuelle sous le répertoire `plugins/` (il faut aussi que le développeur ait envoyé un package)
- La création d'un `svn:externals` dans `plugins/` pour le plugin (il ne fonctionne que si l'hôte du développeur du plugin est un plugin sur Subversion)

Les deux dernières méthodes sont faciles, mais manquent de souplesse. La première manière vous permet d'installer la dernière version selon la version du projet symfony, il est facile de mettre à jour la dernière version stable, et de gérer facilement les dépendances entre les plugins.

La contribution d'un Plugin

Faire le package d'un plugin

Pour créer un package de plugin, vous devez ajouter quelques fichiers obligatoires à la structure du répertoire du plugin. D'abord, créez un fichier README à la racine du répertoire de plugin, et expliquez comment installer le plugin, ce qu'il fait et ce qu'il fait pas. Le fichier README doit être formaté avec le format Markdown. Ce fichier sera utilisé sur le site symfony comme la pièce principale de la documentation. Vous pouvez tester la conversion de votre fichier README au format HTML en utilisant le plugin dingus de symfony.

Tâches de développement d'un plugin

Si vous devez créer souvent des plugins privés et/ou publiques, envisagez d'utiliser certaines des tâches dans le sfTaskExtraPlugin. Ce plugin, maintenu par l'équipe de base, comprend un certain nombre de tâches qui vous aident à rationaliser le cycle de vie du plugin :

- `generate:plugin`

- plugin:package

Vous devez également créer un fichier `LICENSE`. Le choix d'une licence n'est pas une tâche facile, mais la section plugin de symfony répertorie uniquement les plugins qui sont publiées sous une licence similaire à celle symfony (MIT, BSD, LGPL, et PHP). Le contenu du fichier `LICENSE` sera affiché sous l'onglet licence de la page publique de votre plugin.

La dernière étape est de créer un fichier `package.xml` à la racine du répertoire du plugin. Ce fichier `package.xml` suit la syntaxe d'un paquet PEAR.

La meilleure façon d'apprendre la syntaxe du `package.xml` est certainement de copier celui qui est utilisé par un plugin existant.

Le fichier `package.xml` est composé de plusieurs parties comme vous pouvez le voir dans ce modèle par exemple :

```
<!-- plugins/sfJobeetPlugin/package.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<package packagerversion="1.4.1" version="2.0"
   xmlns="http://pear.php.net/dtd/package-2.0"
   xmlns:tasks="http://pear.php.net/dtd/tasks-1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://pear.php.net/dtd/tasks-1.0
   http://pear.php.net/dtd/tasks-1.0.xsd http://pear.php.net/dtd/package-2.0
   http://pear.php.net/dtd/package-2.0.xsd"
>
  <name>sfJobeetPlugin</name>
  <channel>plugins.symfony-project.org</channel>
  <summary>A job board plugin.</summary>
  <description>A job board plugin.</description>
  <lead>
    <name>Fabien POTENCIER</name>
    <user>fabpot</user>
    <email>fabien.potencier@symfony-project.com</email>
    <active>yes</active>
  </lead>
  <date>2008-12-20</date>
  <version>
    <release>1.0.0</release>
    <api>1.0.0</api>
  </version>
  <stability>
    <release>stable</release>
    <api>stable</api>
  </stability>
  <license uri="http://www.symfony-project.com/license">
    MIT license
  </license>
  <notes />

  <contents>
    <!-- CONTENT -->
  </contents>

  <dependencies>
   <!-- DEPENDENCIES -->
  </dependencies>

  <phprelease>
</phprelease>

<changelog>
  <!-- CHANGELOG -->
</changelog>
</package>
```

La balise `<contents>` contient les fichiers qui doivent être placés dans le paquet :

```
<contents>
  <dir name="/">
```

```
    <file role="data" name="README" />
    <file role="data" name="LICENSE" />

    <dir name="config">
      <file role="data" name="config.php" />
      <file role="data" name="schema.yml" />
    </dir>

    <!-- ... -->
  </dir>
</contents>
```

La balise <dependencies> référence toutes les dépendances du plugin qu'il pourrait y avoir : PHP, symfony et aussi d'autres plugins. Cette information est utilisée par la tâche `plugin:install` pour installer la meilleure version du plugin pour l'environnement du projet et d'installer aussi les dépendances requises du plugin le cas échéant.

```
<dependencies>
  <required>
    <php>
      <min>5.0.0</min>
    </php>
    <pearinstaller>
      <min>1.4.1</min>
    </pearinstaller>
    <package>
      <name>symfony</name>
      <channel>pear.symfony-project.com</channel>
      <min>1.3.0</min>
      <max>1.5.0</max>
      <exclude>1.5.0</exclude>
    </package>
  </required>
</dependencies>
```

Vous devez toujours déclarer une dépendance sur symfony, comme nous l'avons fait ici. La déclaration d'une version minimum et maximum permet à `plugin:install` de savoir quelle version de symfony est obligatoire car les versions de symfony peuvent avoir de légères différences dans l'API.

La déclaration d'une dépendance avec un autre plugin est aussi possible :

```
<package>
  <name>sfFooPlugin</name>
  <channel>plugins.symfony-project.org</channel>
  <min>1.0.0</min>
  <max>1.2.0</max>
  <exclude>1.2.0</exclude>
</package>
```

La balise <changelog> est facultative, mais donne des informations utiles sur ce qui a changé entre les versions. Cette information est disponible sous l'onglet "Changelog" et aussi dans le flux du plugin.

```
<changelog>
  <release>
    <version>
      <release>1.0.0</release>
      <api>1.0.0</api>
    </version>
    <stability>
      <release>stable</release>
      <api>stable</api>
    </stability>
    <license uri="http://www.symfony-project.com/license">
      MIT license
    </license>
    <date>2008-12-20</date>
    <license>MIT</license>
    <notes>
       * fabien: First release of the plugin
    </notes>
```

```
  </release>
</changelog>
```

Hébergement d'un plugin sur le site web de symfony

Si vous développez un plugin utile et vous souhaitez le partager avec la communauté de symfony, créez un compte symfony si vous n'en avez pas déjà un, puis créer un nouveau plugin.

Vous deviendrez automatiquement l'administrateur pour le plugin et vous verrez un onglet "admin" dans l'interface. Dans cet onglet, vous trouverez tout ce dont vous avez besoin pour gérer votre plugin et téléchargez vos paquets.

Le FAQ du plugin contient beaucoup d'informations utiles pour les développeurs de plugin.

À demain

Créer des plugins et les partager avec la communauté est l'une des meilleures façons de contribuer en retour au projet symfony. Cela est si facile, que le dépôt de plugin de symfony est rempli de plugins utiles, fun, mais aussi de plugins ridicules.

« Jour 19 : Internationalisation et régionalisation

Jour 21 : Le Cache »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 21 : Le Cache

Aujourd'hui, nous allons parler de la mise en cache. Le framework symfony a beaucoup de stratégies intégrées du cache. Par exemple, les fichiers de configuration YAML sont d'abord converties en PHP et mis en cache sur le système de fichiers. Nous avons également vu que les modules générés par l'admin generator sont mis en cache pour une meilleure performance.

Mais aujourd'hui, nous allons parler d'une autre cache : le cache HTML. Pour améliorer les performances de votre site, vous pouvez mettre en cache toutes les pages HTML ou juste une partie d'entre elles.

La création d'un nouvel environnement

Par défaut, la fonctionnalité du cache de template de symfony est activée dans le fichier de configuration `settings.yml` pour l'environnement de `prod`, mais pas pour celui de `test` et `dev` :

```
prod:
  .settings:
    cache: true

dev:
  .settings:
    cache: false

test:
  .settings:
    cache: false
```

Comme nous avons besoin de tester la fonctionnalité du cache avant d'aller en production, nous pouvons activer le cache pour l'environnement de dev ou créer un nouvel environnement. Rappelons qu'un environnement est défini par son nom (une chaîne), un contrôleur frontal et éventuellement un ensemble de valeur de configuration spécifique.

Pour jouer avec le système de cache sur Jobeet, nous allons créer un environnement cache, similaire à l'environnement `prod`, mais avec la journalisation et les informations de débogage disponibles dans l'environnement de `dev`.

Créez le contrôleur frontal associé au nouvel environnement cache en copiant le contrôleur frontal de devweb/frontend_dev.phpversweb/frontend_cache.php` :

```
// web/frontend_cache.php
if (!in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', '::1')))
{
  die('You are not allowed to access this file. Check '.basename(__FILE__).' for more
information.');
}

require_once(dirname(__FILE__).'/../config/ProjectConfiguration.class.php');

$configuration = ProjectConfiguration::getApplicationConfiguration('frontend',
'cache', true);
sfContext::createInstance($configuration)->dispatch();
```

C'est tout ce qu'il y a à faire. Le nouvel environnement cache est maintenant utilisable. La seule différence est que le deuxième argument de la méthode `getApplicationConfiguration()` est le nom de l'environnement : `cache`.

Vous pouvez tester l'environnement cache dans votre navigateur en appelant son contrôleur frontal :

```
http://www.jobeet.com.localhost/frontend_cache.php/
```

Le script du contrôleur frontal commence par un code qui assure que le contrôleur frontal est seulement appelé à partir d'une adresse IP locale. Cette mesure de sécurité permet d'éviter que le contrôleur frontal soit appelé sur les serveurs de production. Nous en reparlerons plus en détail dans le tutoriel de demain.

Pour l'instant, l'environnement cache hérite de la configuration par défaut. Modifiez le fichier de configuration `settings.yml` pour ajouter à l'environnement cache une configuration spécifique :

```
# apps/frontend/config/settings.yml
cache:
  .settings:
    error_reporting: <?php echo (E_ALL | E_STRICT)."\n" ?>
    web_debug:       true
    cache:           true
    etag:            false
```

Dans ces paramètres, la fonctionnalité du cache de template de symfony a été activée avec le paramètre cache et le web debug toolbar a été activé avec le paramètre web_debug.

Comme la configuration par défaut met en cache tous les paramètres, vous avez besoin de le vider avant d'être en mesure de voir les changements dans votre navigateur :

```
$ php symfony cc
```

Maintenant, si vous actualisez votre navigateur, le web debug toolbar devrait être présent dans le coin en haut à droite de la page, comme c'est le cas pour l'environnement dev.

Configuration du cache

Le cache du template de symfony peut être configuré avec le fichier de configuration `cache.yml`. La configuration par défaut pour l'application se trouve dans `apps/frontend/config/cache.yml` :

```
default:
  enabled:     false
  with_layout: false
  lifetime:    86400
```

Par défaut, comme toutes les pages peuvent contenir des informations dynamiques, le cache est désactivé totalement (`enabled: false`). Nous n'avons pas besoin de modifier ce paramètre, car nous allons activer le cache page par page.

Le paramètre `lifetime` définit la durée de vie du cache du côté du serveur en secondes (`86400` secondes est égal à un jour).

Vous pouvez également travailler dans l'autre sens : activez le cache globalement, puis, le désactivez sur des pages spécifiques qui ne peuvent être mises en cache. Cela dépend du choix représentant le moins de travail pour votre application.

Mise en cache d'une page

Comme la page d'accueil Jobeet sera probablement la page la plus visitée du site web, au lieu de requêter les données dans la base de données chaque fois qu'un utilisateur accède à la page, elle peut être mis en cache.

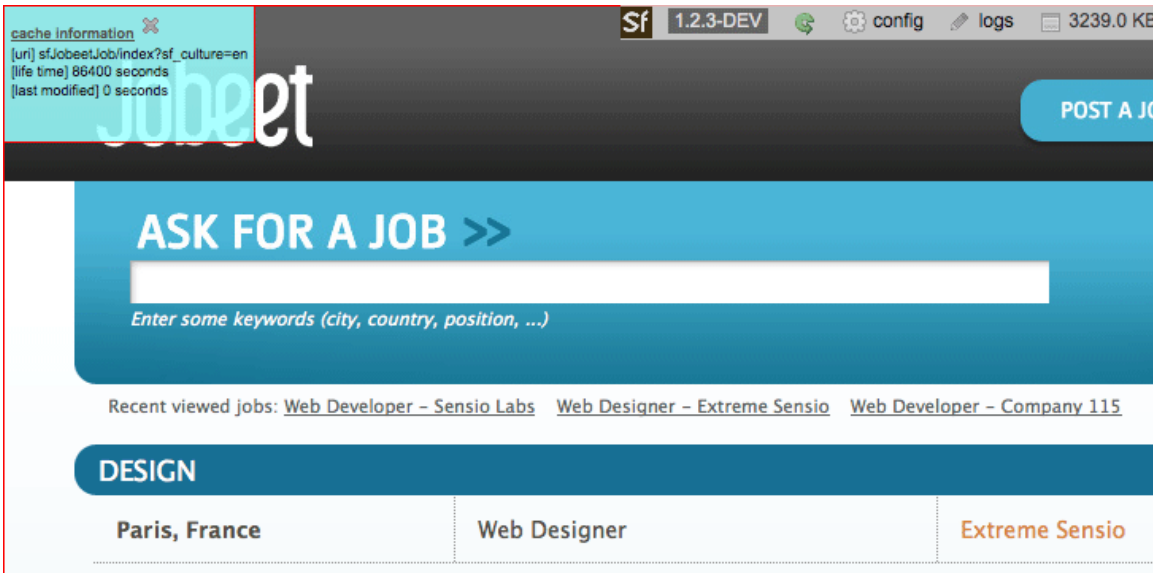
Créez un fichier `cache.yml` pour le module `sfJobeetJob` :

```
# plugins/sfJobeetPlugin/modules/sfJobeetJob/config/cache.yml
index:
  enabled:     true
  with_layout: true
```

Le fichier de configuration `cache.yml` a les mêmes propriétés que tous les autres fichiers de configuration de symfony comme `view.yml`. Cela signifie par exemple que vous pouvez activer le cache pour toutes les actions d'un module en utilisant la clé spéciale `all`.

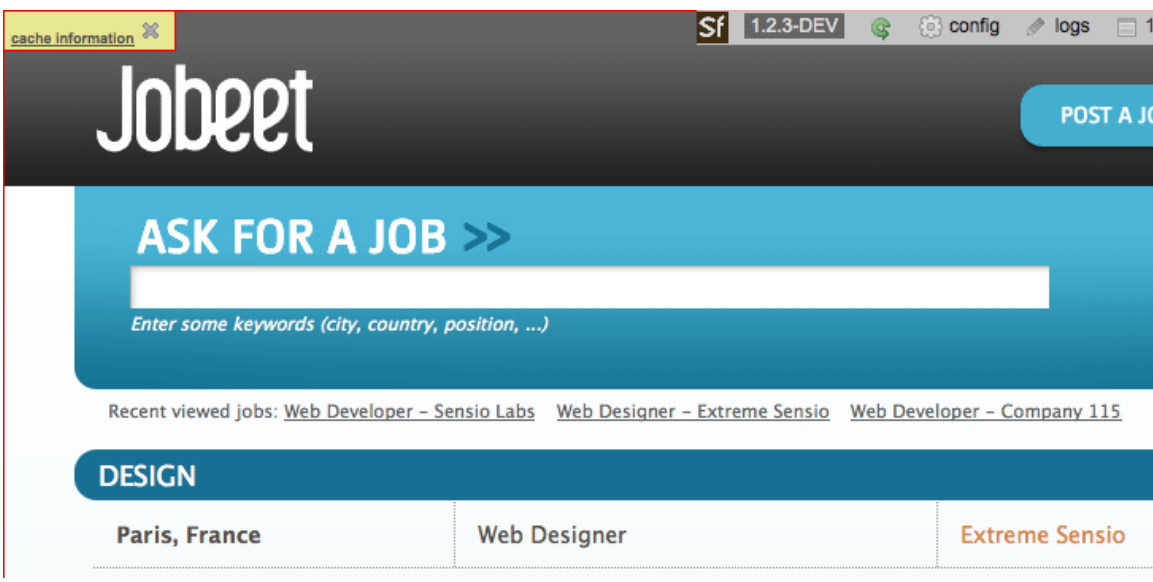
Si vous actualisez votre navigateur, vous pourrez voir que symfony a décoré la page avec une

case indiquant que le contenut a été mise en cache :



La boîte donne des informations précieuses sur la clé du cache pour le débogage, comme la durée de vie du cache, et l'âge de celui-ci.

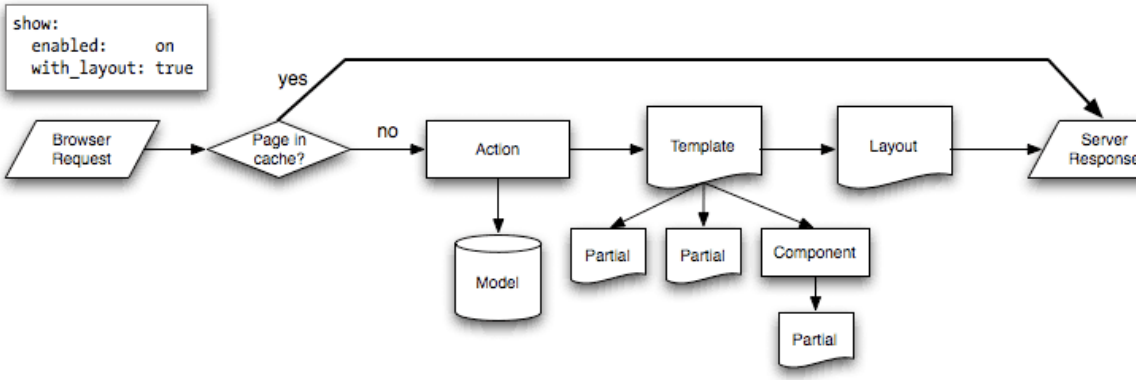
Si vous actualisez la page à nouveau, la couleur de la boîte passe du vert au jaune, indiquant que la page a été extraites de la mémoire cache :



Notez également qu'aucune demande de base de données a été faite dans le second cas, comme indiqué dans le web debug toolbar.

Même si la langue peut être changée selon l'utilisateur, le cache fonctionne encore car la langue est incorporé dans l'URL.

Quand une page est mis en cache, et si le cache n'existe pas encore, symfony stocke l'objet de la réponse dans le cache à la fin de la requête. Pour toutes les autres requêtes futures, symfony va envoyer la réponse mise en cache sans appeler le contrôleur :



Cela a un impact considérable sur la performance que vous pouvez mesurer par vous-même en utilisant des outils tels que JMeter.

Une requête entrante avec des paramètres GET ou soumis avec la méthode POST, PUT ou DELETE ne sera jamais mis en cache par symfony, quelle que soit la configuration.

La page de création d'emplois peut aussi être mis en cache :

```
# plugins/sfJobeetPlugin/modules/sfJobeetJob/config/cache.yml
new:
  enabled:     true

index:
  enabled:     true

all:
  with_layout: true
```

Comme les deux pages peuvent être mises en cache avec le layout, nous avons créé une section `all` qui définit la configuration par défaut pour l'ensemble des actions du module `sfJobeetJob`.

Le vidage du cache

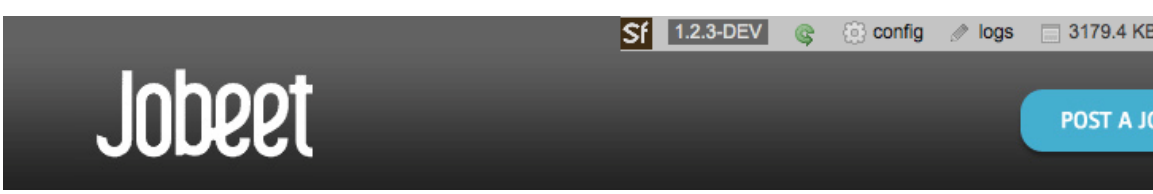
Si vous voulez vider le cache de la page, vous pouvez utiliser la tâche `cache:clear` :

```
$ php symfony cc
```

La tâche `cache:clear` vide tous les caches stockés sous le répertoire principal `cache/`. Il prend également des options pour vider sélectivement certaines parties du cache. Pour vider uniquement le cache de template pour l'environnement de `cache`, utilisez les options `--type` et `--env` :

```
$ php symfony cc --type=template --env=cache
```

Au lieu d'effacer le cache chaque fois que vous apportez une modification, vous pouvez également désactiver le cache en ajoutant toute chaîne de requête à l'URL, ou en utilisant le bouton "Ignore cache" depuis le web debug toolbar :



Mise en cache d'une action

Parfois, vous ne pouvez pas mettre en cache la page entière dans le cache, mais seule l'action du template peut être mis en cache. Autrement dit, vous pouvez tout mettre en cache, sauf le layout.

Pour l'application Jobeet, nous ne pouvons pas mettre en cache la page entière à cause de la barre "history job".

Modifiez la configuration le cache pour le module `job` en conséquence :

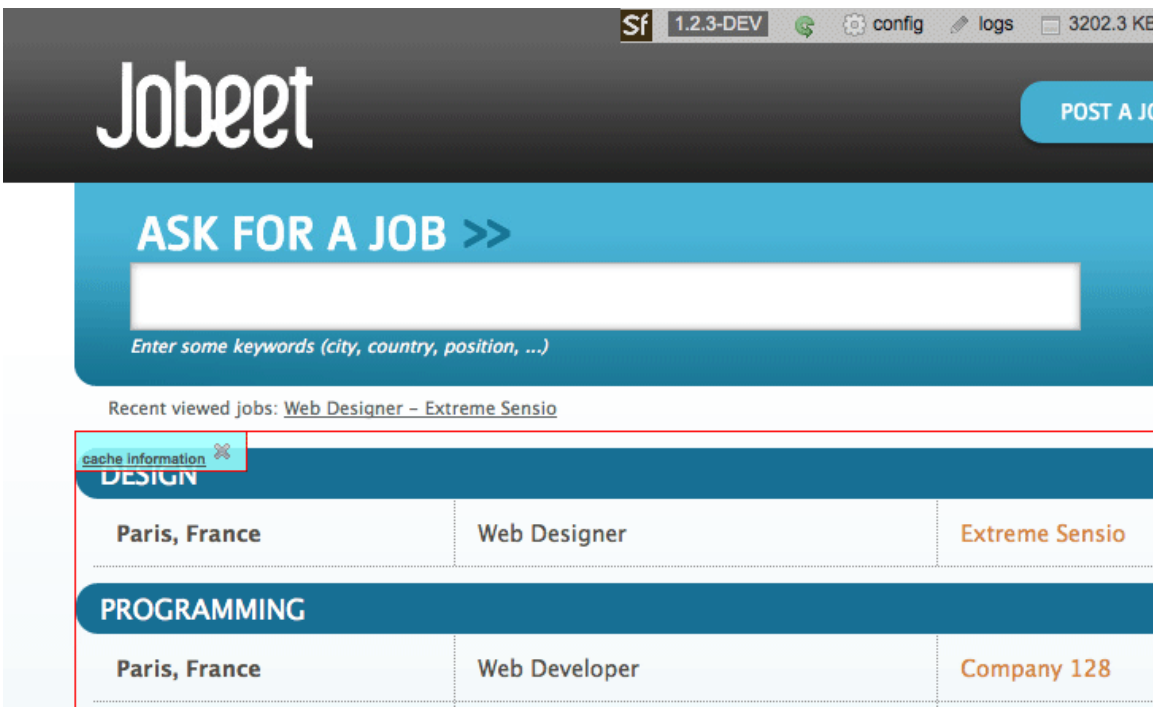
```
# plugins/sfJobeetPlugin/modules/sfJobeetJob/config/cache.yml
new:
  enabled:     true

index:
  enabled:     true

all:
  with_layout: false
```

En changeant le paramètre `with_layout` à `false`, vous avez désactivé la mise en cache du layout.

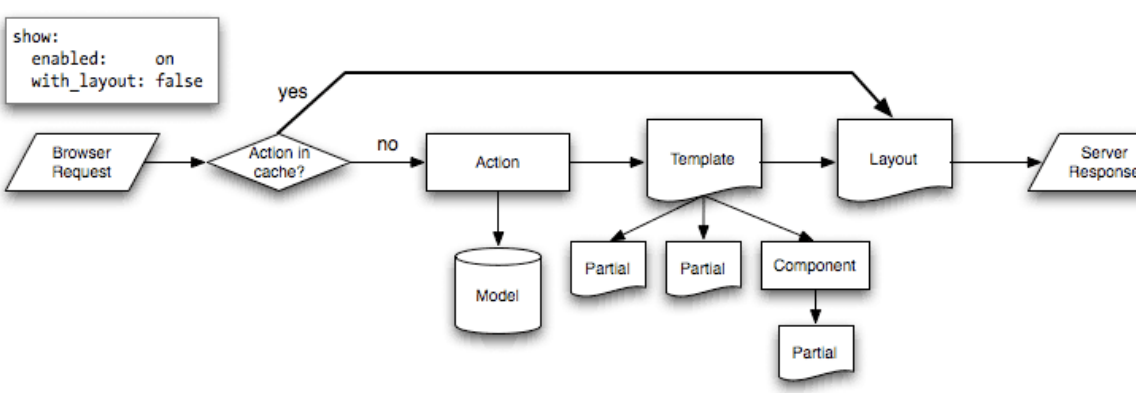
Vider le cache:

```
$ php symfony cc
```

Rafraîchissez votre navigateur pour voir la différence :



Même si le flux de la requête est assez similaire dans le diagramme simplifié, la mise en cache sans le layout est la plus intensive ressource.



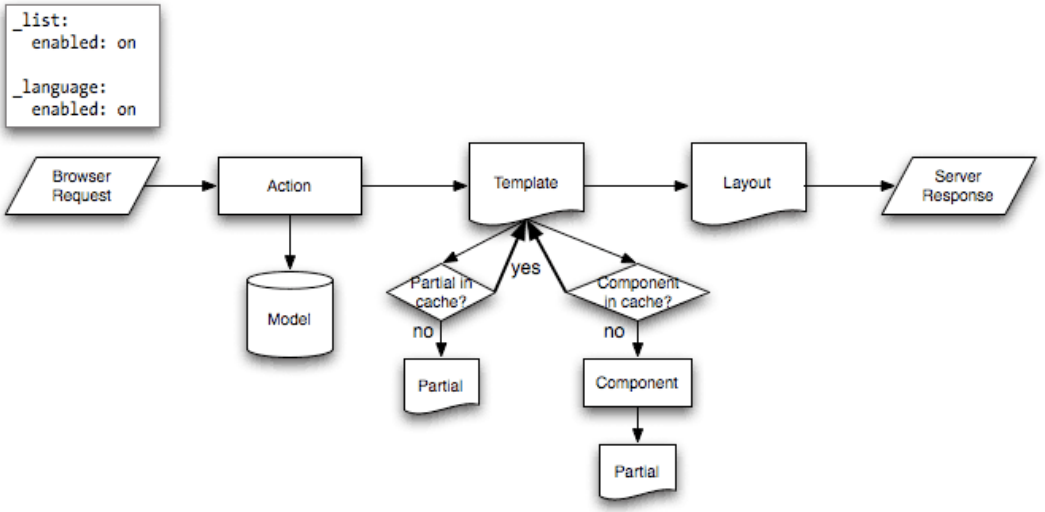
Mise en cache du Partial et du Component

Pour les sites web très dynamique, il est parfois même impossible de mettre en cache toute l'action du template. Pour ces cas, il vous faut un moyen de configurer le cache au niveau le plus fin. Heureusement, les partials et les components peuvent également être mis en cache.

Mettons en cache le component `language` en créant un fichier `cache.yml` pour le module `sfJobeetLanguage` :

```
# plugins/sfJobeetPlugin/modules/sfJobeetLanguage/config/cache.yml
_language:
  enabled: true
```

La configuration du cache d'un partial ou d'un component est aussi simple que l'ajout d'une entrée avec son nom. L'option `with_layout` n'est pas pris en compte pour ce type de cache car cela n'a pas de sens :



Contextuel ou non ?

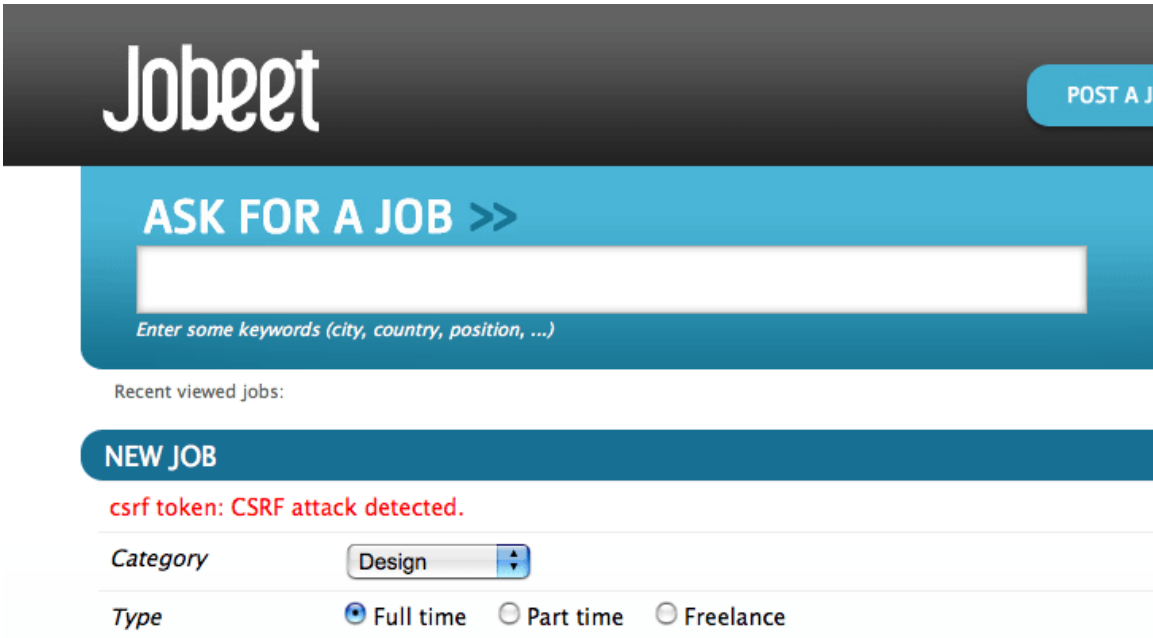
La même component ou partial peut être utilisé dans beaucoup de templates différents. Le partial d'emploi `_list.php`, par exemple, est utilisé dans les modules `sfJobeetJob` et `sfJobeetCategory`. Comme le rendu est toujours le même, le partial ne dépend pas du contexte dans lequel il est utilisé et le cache est le même pour tous les templates (le cache est bien évidemment différent pour une autre série de paramètres).

Mais parfois, l'affichage d'un partial ou d'un component est différent, selon l'action dans laquel il est inclus (pensez à une barre latérale d'un blog par exemple, qui est légèrement différente pour la page d'accueil et la page d'un message du blog). Dans ces cas, le partial ou le component est contextuel, et le cache doit être configuré en conséquence en mettant l'option `contextual` à `true` :

```
_sidebar:
  enabled:    true
  contextual: true
```

Formulaires en cache

Le stockage de la page de création d'emplois dans le cache est problématique, car il contient un formulaire. Pour mieux comprendre le problème, allez à la page "Post a Job" dans votre navigateur pour voir le cache. Ensuite, désactivez votre cookie de session, et essayer de soumettre un emploi. Vous devez voir un message d'erreur vous avertissant d'une "attaque CSRF" :



Pourquoi? Comme nous avons configuré un secret CSRF lorsque nous avons créé l'application frontend, symfony intègre un jeton CSRF dans toutes ses formulaires. Afin de vous protéger contre les attaques CSRF, ce jeton est unique pour un utilisateur donné et pour un formulaire donné.

La première fois que la page est affichée, le formulaire HTML généré est stocké dans le cache avec le jeton de l'utilisateur en cours. Si un autre utilisateur vient après, la page du cache sera affichée avec le jeton CSRF du premier utilisateur. En soumettant le formulaire, les jetons ne correspondent pas et une erreur est levée.

Comment pouvons-nous résoudre le problème car il semble légitime de stocker le formulaire dans le cache ? Le formulaire de création d'emplois ne dépend pas de l'utilisateur, et cela ne change rien pour l'utilisateur actuel. Dans un tel cas aucune protection CSRF est nécessaire, et nous pouvons éliminer le CSRF jeton :

```
// plugins/sfJobeetPlugin/lib/form/doctrine/PluginJobeetJobForm.class.php
abstract PluginJobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    $this->disableLocalCSRFProtection();
  }
}
```

Après avoir fait ce changement, videz le cache et ré-essayez le même scénario que ci-dessus pour prouver qu'il fonctionne comme prévu maintenant.

La même configuration doit être appliquée au formulaire de la langue qui est contenu dans le layout et qui sera stockées dans le cache. Comme le `sfLanguageForm` par défaut est utilisée, au lieu de créer une nouvelle classe, il suffit de supprimer le jeton CSRF, faisons-le depuis l'action et le component du module `sfJobeetLanguage` :

```
// plugins/sfJobeetPlugin/modules/sfJobeetLanguage/actions/components.class.php
class sfJobeetLanguageComponents extends sfComponents
{
  public function executeLanguage(sfWebRequest $request)
  {
    $this->form = new sfFormLanguage($this->getUser(), array('languages' =>
array('en', 'fr')));
    $this->form->disableLocalCSRFProtection();
  }
}

// plugins/sfJobeetPlugin/modules/sfJobeetLanguage/actions/actions.class.php
class sfJobeetLanguageActions extends sfActions
{
  public function executeChangeLanguage(sfWebRequest $request)
  {
```

```
    $form = new sfFormLanguage($this->getUser(), array('languages' => array('en',
'fr')));
    $form->disableLocalCSRFProtection();

    // ...
  }
}
```

La méthode `disableLocalCSRFProtection()` désactive le jeton unique CSRF du formulaire courant.

Suppression du cache

Chaque fois qu'un utilisateur poste et active un emploi, la page d'accueil doit être rafraîchie pour lister le nouvel emploi.

Comme nous n'avons pas besoin que l'emploi apparaisse en temps réel sur la page d'accueil, la meilleure stratégie est de diminuer la durée de vie du cache vers quelque chose d'acceptable :

```
# plugins/sfJobeetPlugin/modules/sfJobeetJob/config/cache.yml
index:
  enabled:  true
  lifetime: 600
```

Au lieu que la configuration par défaut soit d'un jour, le cache de la page d'accueil sera supprimée automatiquement toutes les dix minutes.

Mais si vous voulez mettre à jour la page d'accueil dès que l'utilisateur active un nouvel emploi, éditez la méthode `executePublish()` du module `sfJobeetJob` pour ajouter le vidage du cache manuellement :

```
// plugins/sfJobeetPlugin/modules/sfJobeetJob/actions/actions.class.php
public function executePublish(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $job->publish();

  if ($cache = $this->getContext()->getViewCacheManager())
  {
    $cache->remove('sfJobeetJob/index?sf_culture=*');
    $cache->remove('sfJobeetCategory/show?id='.$job->getJobeetCategory()->getId());
  }

  $this->getUser()->setFlash('notice', sprintf('Your job is now online for %s days.',
sfConfig::get('app_active_days')));

  $this->redirect($this->generateUrl('job_show_user', $job));
}
```

Le cache est géré par la classe `sfViewCacheManager`. La méthode `remove()` supprime le cache associé à une URI interne. Pour retirer le cache pour tous les paramètres possibles d'une variable, utilisez le `*` en tant que valeur. Le `sf_culture=*` que nous avons utilisé dans le code ci-dessus signifie que symfony va supprimer le cache pour les pages d'accueil anglaise et française.

Come le gestionnaire de cache est `null` lorsque le cache est désactivé, nous avons enveloppé la suppression du cache dans un bloc `if`.

Test du cache

Avant de commencer, nous devons changer la configuration de l'environnement `test` pour activer la couche du cache :

```
# apps/frontend/config/settings.yml
test:
  .settings:
    error_reporting: <?php echo ((E_ALL | E_STRICT) ^ E_NOTICE)."\n" ?>
    cache:           true
    web_debug:       false
    etag:            false
```

Testons la page de création d'emplois :

```
// test/functional/frontend/jobActionsTest.php
$browser->
  info('  7 - Job creation page')->

  get('/fr/')->
  with('view_cache')->isCached(true, false)->

  createJob(array('category_id' => Doctrine_Core::getTable('JobeetCategory')->
>findOneBySlug('programming')->getId()), true)->

  get('/fr/')->
  with('view_cache')->isCached(true, false)->
  with('response')->checkElement('.category_programming .more_jobs', '/23/')
;
```

Le testeur `view_cache` est utilisé pour tester le cache. La méthode `isCached()` accepte deux booléens :

- Savoir si la page doit être dans le cache ou pas
- Savoir si le cache est avec un layout ou non

Même avec tous les outils fournis par le framework de test fonctionnel, il est parfois plus facile de diagnostiquer les problèmes au sein du navigateur. Il est assez facile à accomplir. Il suffit de créer un contrôleur frontal pour l'environnement `test`. Les logs qui sont stockés dans `log/frontend_test.log` peuvent aussi être très utile.

À demain

Comme beaucoup d'autres fonctionnalités symfony, le sous-framework de cache de symfony est très souple et permet au développeur de configurer le cache à un niveau très fin.

Demain, nous allons parler de la dernière étape du cycle de vie de l'application : le déploiement des serveurs de production.

« Jour 20 : Les Plugins

Jour 22 : Le déploiement »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Practical symfony
Jour 22 : Le déploiement

Hier, avec la configuration du système de cache, le site Jobeet est prêt à être déployé sur les serveurs de production.

Pendant vingt-deux jours, nous avons développé Jobeet sur une machine de développement, et pour la plupart d'entre vous, cela signifie probablement sur votre machine locale; sauf si vous développez sur le serveur de production directement, ce qui est évidemment une très mauvaise idée. Maintenant, il est temps de passer le site vers un serveur de production.

Aujourd'hui, nous allons voir ce qui doit être fait avant d'aller en production, quel genre de stratégies de déploiement pouvez vous utiliser, et aussi les outils dont vous avez besoin pour un déploiement réussi.

Préparation du serveur de production

Avant de déployer le projet en production, nous devons être sûr que le serveur de production est correctement configuré. Vous pouvez relire le jour 1, où nous avons expliqué comment configurer le serveur web.

Dans cette section, nous supposons que vous avez déjà installé le serveur web, le serveur de base de données et PHP 5.2.4 ou ultérieur.

Si vous ne disposez pas d'un accès SSH au serveur web, ignorez la partie où vous avez besoin d'avoir accès à la ligne de commande.

Configuration du serveur

D'abord, vous devez vérifier que PHP est installé avec toutes les extensions nécessaires et qu'il est correctement configuré. Comme pour le jour 1, nous allons utiliser le script fourni `check_configuration.php` avec symfony. Comme nous ne voulons pas installer symfony sur le serveur de production, téléchargez le fichier directement sur le site de symfony :

```
http://trac.symfony-project.org/browser/branches/1.4/data/bin/check_configuration.php?format=raw
```

Copiez le fichier dans le répertoire racine Web et exécutez-le depuis votre navigateur **et** en ligne de commande :

```
$ php check_configuration.php
```

Corrigez les erreurs fatales que le script trouve et répétez le processus jusqu'à ce que tout fonctionne bien dans les **deux** environnements.

Accelerateur PHP

Pour le serveur de production, vous voudrez probablement la meilleure performance possible. L'installation d'un accélérateur PHP vous donnera la meilleure amélioration possible.

Extrait de Wikipedia: Un accélérateur PHP fonctionne en mettant en cache le code compilé des scripts PHP pour éviter le surcoût de l'analyse et la compilation du code source sur chaque requête.

APC est l'un des plus populaire et il est assez simple à l'installer :

```
$ pecl install APC
```

En fonction de votre système d'exploitation, vous serez également capable de l'installer avec le

gestionnaire de paquet natif de l'OS.

Prenez le temps d'apprendre à configurer APC.

Les bibliothèques de symfony

Intégration de symfony

Une des grandes forces de symfony est que le projet est auto-contenue. Tous les fichiers nécessaires pour que le projet fonctionne, sont sous le répertoire principal racine du projet. Et vous pouvez déplacer le projet dans un autre répertoire sans rien changer au projet lui-même car symfony n'utilise que des chemins relatifs. Cela signifie que le répertoire sur le serveur de production ne doit pas être le même que celui de votre machine de développement.

Le seul chemin absolu qui peut éventuellement être trouvé est dans le fichier `config/ProjectConfiguration.class.php`, mais nous avons pris soin de cela pendant la journée 1. Assurez-vous qu'elle contient en fait un chemin relatif vers le chargeur automatique du noyau de symfony :

```
// config/ProjectConfiguration.class.php
require_once
dirname(__FILE__).'/../lib/vendor/symfony/lib/autoload/sfCoreAutoload.class.php';
```

Mise à jour de symfony

Même si tout est auto-contenue dans un seul répertoire, la mise à niveau de symfony pour une version plus récente est pourtant incroyablement facile.

Vous souhaitez mettre à niveau symfony vers la dernière version mineure de temps en temps, comme nous corrigeons en permanence des bugs et éventuellement, les questions de sécurité. Les bonnes nouvelles sont que toutes les versions de symfony sont maintenues pendant au moins un an et pendant la période de maintenance, nous n'ajoutons jamais de nouvelles fonctionnalités, même les plus petites. Ainsi, il est toujours rapide, sûr et sécurisé pour passer d'une version mineure à une autre.

La mise à jour de symfony est aussi simple que de changer le contenu du répertoire `lib/vendor/symfony/`. Si vous avez installé symfony avec l'archive, supprimez les fichiers actuels et remplacez les par les plus récents.

Si vous utilisez Subversion pour votre projet, vous pouvez aussi lier votre projet au dernièr tag de symfony 1.4 :

```
$ svn propedit svn:externals lib/vendor/
  # symfony http://svn.symfony-project.com/tags/RELEASE_1_4_0/
```

La mise à jour de symfony est donc aussi simple que de changer le tag de la dernière version de symfony.

Vous pouvez également utiliser la branche 1.4 pour corriger en temps réel :

```
$ svn propedit svn:externals lib/vendor/
  # symfony http://svn.symfony-project.com/branches/1.4/
```

Maintenant, chaque fois que vous faites un `svn up`, vous avez la dernière version 1.4 de symfony.

Lors de la mise à jour d'une nouvelle version, il vous est conseillé de toujours vider le cache, en particulier dans l'environnement de production :

```
$ php symfony cc
```

Si vous avez aussi un accès FTP au serveur de production, vous pouvez simuler un `symfony cc` simplement en enlevant tous les fichiers et répertoires sous le répertoire `cache/`.

Vous pouvez même tester une nouvelle version de symfony, sans remplacer l'existant. Si vous voulez juste tester une nouvelle version, et nous voulons être en mesure de restaurer facilement, installer symfony dans un autre répertoire (`lib/vendor/symfony_test` par exemple), changer le chemin dans la classe `ProjectConfiguration`, videz le cache et c'est fini . Le retour arrière est aussi simple que la suppression du répertoire et le changement du chemin dans `ProjectConfiguration`.

Peaufiner la configuration

Configuration de la base de données

La plupart du temps, la base de données de production a des droits différents de ceux en local. Grâce aux environnements de symfony, il est assez simple d'avoir une configuration différente pour la base de données de production :

```
$ php symfony configure:database "mysql:host=localhost;dbname=prod_dbname" prod_user
prod_pass
```

Vous pouvez également modifier le fichier de configuration `databases.yml` directement.

Ressources

Comme Jobeet utilise des plugins qui intègrent des ressources, symfony crée des liens symboliques relatifs dans le répertoire `web/`. La tâche `plugin:publish-assets` les régénère ou les crée si vous installer des plugins sans la tâche `plugin:install` :

```
$ php symfony plugin:publish-assets
```

Personnalisation des Pages d'erreur

Avant d'aller en production, il est préférable de personnaliser les pages de symfony par défaut, comme la "Page introuvable" ou la page d'exceptions par défaut.

Nous avons déjà configuré la page d'erreur pour le format YAML pendant le jour 15, en créant les fichiers `error.yaml.php` et `exception.yaml.php` dans le répertoire `config/error/`. Le fichier `error.yaml.php` est utilisé dans l'environnement `prod`, alors que `exception.yaml.php` est utilisé dans l'environnement de `dev`.

Donc, pour personnaliser la page d'exception par défaut pour le format HTML, créez deux fichiers : `config/error/error.html.php` et `config/error/exception.html.php`.

La page 404 (page introuvable) peut être personnalisée en changeant les paramètres `error_404_module` et `error_404_action` :

```
# apps/frontend/config/settings.yml
all:
  .actions:
    error_404_module: default
    error_404_action: error404
```

Personnalisation de la structure de répertoire

Afin de mieux structurer et de normaliser votre code, symfony a une structure de répertoire par défaut avec des noms prédéfinis. Mais parfois, vous n'avez pas le choix, il faut changer la structure à cause de certaines contraintes extérieures.

La configuration des noms de répertoire peut être fait dans la classe `config/ProjectConfiguration.class.php`.

Le répertoire racine web

Chez certains hébergeurs, vous ne pouvez pas changer le nom du répertoire racine web. Disons que chez votre hébergeur, il est nommé `public_html/` au lieu de `web/` :

```
// config/ProjectConfiguration.class.php
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    $this->setWebDir($this->getRootDir().'/public_html');
  }
}
```

La méthode `setWebDir()`prend le chemin absolu du répertoire racine web. Si vous déplacez aussi ce répertoire ailleurs, n'oubliez pas de modifier les scripts du contrôleur pour vérifier que les chemins du fichier `config/ProjectConfiguration.class.php` sont toujours valables :

```
require_once(dirname(__FILE__).'/../config/ProjectConfiguration.class.php');
```

Les répertoires Cache et Log

Le framework Symfony écrit que dans deux répertoires: `cache/` et `log/`. Pour des raisons de

sécurité, certains hébergeurs n'acceptent pas les permissions en écriture dans le répertoire principal. Si tel est le cas, vous pouvez déplacer ces répertoires ailleurs sur le système de fichiers :

```
// config/ProjectConfiguration.class.php
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    $this->setCacheDir('/tmp/symfony_cache');
    $this->setLogDir('/tmp/symfony_logs');
  }
}
```

Comme pour la méthode `setWebDir()`, `setCacheDir()` et `setLogDir()` prennent un chemin absolu pour les répertoires respectifs `cache/` et `log/`.

Personnalisation des objets du noyau de symfony (via les factories)

Pendant le jour 16, nous avons un peu parlé des factories de symfony. Être en mesure de personnaliser les factories, signifie que vous pouvez utiliser une classe personnalisée pour les objets du noyau de symfony à la place de celui par défaut. Vous pouvez également modifier le comportement par défaut de ces classes en changeant les paramètres qui leurs sont envoyés.

Jetons un oeil sur quelques personnalisations classisue que vous pouvez faire.

Nom du Cookie

Pour gérer la session utilisateur, symfony utilise un cookie. Ce cookie a un nom par défaut `symfony`, qui peut être changé dans `factories.yml`. Sous la clé `all`, ajoutez la configuration suivante pour changer le nom du cookie en `jobeet` :

```
# apps/frontend/config/factories.yml
storage:
  class: sfSessionStorage
  param:
    session_name: jobeet
```

Stockage de Session

La classe de session par défaut de stockage est `sfSessionStorage`. Ceci utilise le système de fichiers pour stocker les informations de session. Si vous avez plusieurs serveurs web, vous souhaitez stocker les sessions dans un endroit central, comme une table de base de données :

```
# apps/frontend/config/factories.yml
storage:
  class: sfPDOSessionStorage
  param:
    session_name: jobeet
    db_table:     session
    database:     doctrine
    db_id_col:    id
    db_data_col:  data
    db_time_col:  time
```

Timeout d'une session

Par défaut, le timeout d'une session utilisateur est `1800` secondes. Ceci peut être modifié en éditant l'entrée `user` :

```
# apps/frontend/config/factories.yml
user:
  class: myUser
  param:
    timeout: 1800
```

Journalisation

Par défaut, il n'y a pas de journalisation dans l'environnement `prod` parce que le nom de la classe du journal est `sfNoLogger` :

```
# apps/frontend/config/factories.yml
prod:
```

```
  logger:
    class:   sfNoLogger
    param:
      level:   err
      loggers: ~
```

Vous pouvez par exemple activer la journalisation du système de fichiers en changeant le nom de la classe ddu journal en `sfFileLogger` :

```
# apps/frontend/config/factories.yml
logger:
  class: sfFileLogger
  param:
    level:   err
    loggers: ~
    file:    %SF_LOG_DIR%/%SF_APP%_%SF_ENVIRONMENT%.log
```

Dans le fichier de configuration `factories.yml`, les chaines `%XXX%` sont remplacées par leurs valeurs correspondantes depuis l'objet `sfConfig`. Ainsi, `%SF_APP%` dans un fichier de configuration est équivalent à `sfConfig::get('sf_app')` dans du code PHP. Cette notation peut aussi être utilisé dans le fichier de configuration `app.yml`. Il est très utile lorsque vous avez besoin de faire référence à un chemin dans un fichier de configuration sans coder en dur le chemin (`SF_ROOT_DIR`, `SF_WEB_DIR`, ...).

Déploiement

Ce qu'il faut déployer?

Lors du déploiement du site Jobeet vers le serveur de production, nous devons veiller à ne pas déployer les fichiers inutiles ou remplacer les fichiers téléchargés par nos utilisateurs, comme les logos d'entreprise.

Dans un projet symfony, il existe trois répertoires à exclure du transfert : `cache/`, `log/` et `web/uploads/`. Tout le reste peut être transférée tel quel.

Pour des raisons de sécurité, vous pouvez aussi ne pas vouloir transférer les contrôleurs frontaux de «non-production» , comme les scripts `frontend_dev.php`, `backend_dev.php` et `frontend_cache.php`.

Stratégies de déploiement

Dans cette section, nous supposerons que vous avez le contrôle total sur le(s) serveur(s) de production. Si vous ne pouvez accéder au serveur avec un compte FTP, la seule solution de déploiement possible est de transférer tous les fichiers chaque fois que vous déployez.

Le plus simple pour déployer votre site web est d'utiliser la tâche intégrée `project:deploy`. Elle utilise SSH et `rsync` pour connecter et transférer les fichiers d'un ordinateur à un autre.

Les serveurs pour la tâche `project:deploy` peuvent être configurés dans le fichier de configuration `config/properties.ini` :

```
# config/properties.ini
[production]
  host=www.jobeet.org
  port=22
  user=jobeet
  dir=/var/www/jobeet/
```

Pour déployer sur le serveur nouvellement configuré de `production`, utilisez la tâche `project:deploy` :

```
$ php symfony project:deploy production
```

Avant de lancer la tâche `project:deploy` pour la première fois, vous devez vous connecter manuellement au serveur pour ajouter la clé dans le fichier des host connus.

Si la commande ne fonctionne pas comme prévue, vous pouvez passer l'option `-t` pour voir l'affichage en temps réel de la commande `rsync`.

Si vous exécutez cette commande, symfony va seulement simuler le transfert. Pour déployer réellement le site, ajoutez l'option `--go` :

```
$ php symfony project:deploy production --go
```

Même si vous pouvez fournir le mot de passe SSH dans le fichier `properties.ini`, il est préférable de configurer votre serveur avec une clé SSH pour allouer un mot de pass de connexion.

Par défaut, symfony ne va pas transférer les répertoires dont nous avons parlé dans la section précédente, ni le script du contrôleur frontal du dev. Car la tâche `project:deploy` exclut des fichiers et des répertoires configurés dans le fichier `config/rsync_exclude.txt` :

```
# config/rsync_exclude.txt
.svn
/web/uploads/*
/cache/*
/log/*
/web/*_dev.php
```

Pour Jobeet, nous devons ajouter le fichier `frontend_cache.php` :

```
# config/rsync_exclude.txt
.svn
/web/uploads/*
/cache/*
/log/*
/web/*_dev.php
/web/frontend_cache.php
```

Vous pouvez également créer un fichier `config/rsync_include.txt` pour forcer certains fichiers ou répertoires d'être transférés.

Même si la tâche `project:deploy` est très flexible, vous pouvez encore plus la personnaliser. Comme le déploiement peut être très différent en fonction de votre configuration de serveur et de topologie, n'hésitez pas à étendre la tâche par défaut.

Chaque fois que vous déployez un site web pour la production, n'oubliez pas au moins de vider le cache de configuration sur le serveur de production :

```
$ php symfony cc --type=config
```

Si vous avez modifié certaines routes, vous aurez également besoin de vider le cache de routage :

```
$ php symfony cc --type=routing
```

Vider le cache de manière sélective permet de conserver certaines parties du cache, tels que le cache de template.

À demain

Le déploiement d'un projet est la dernière étape du cycle de vie de développement de symfony. Cela ne signifie pas que vous avez terminé. C'est en fait tout le contraire. Un site web est quelque chose qui a une vie. Vous aurez probablement à corriger les bugs et vous aurez également à ajouter de nouvelles fonctionnalités au fil du temps. Mais grâce à la structure de symfony et les outils à votre disposition, mettre à niveau votre site est simple, rapide et sécurisé.

Demain, c'est le dernier jour du tutoriel Jobeet. Il sera temps de prendre un peu de recul et jetez un œil à ce que vous avez appris au cours des vingt-trois jours de Jobeet.

« Jour 21 : Le Cache

Jour 23 : Un autre regard sur symfony »

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

the SENSIOLABS network

Since 1998, Sensio Labs has been promoting the Open-Source software movement by providing quality web application development, training, consulting.
Sensio Labs also supports several large Open-Source projects.

Open-Source Products

Symfony - MVC framework
Symfony Components
Doctrine - ORM
Swift Mailer - Mailing library
Twig - Template library
Pirum - PEAR channel server

Servic

Training
Guru -
Partner
Books -
Confere
Confere

Practical symfony
Jour 23 : Un autre regard sur symfony

Aujourd'hui est la dernière étape de notre voyage dans le merveilleux monde de symfony. Au cours de ces vingt-trois jours, vous avez appris symfony par exemple du modèle de design utilisés par le framework vers les puissantes fonctionnalités intégrées. Vous n'êtes pas encore un maître de symfony, mais vous avez toutes les connaissances nécessaires pour commencer à construire vos applications symfony avec confiance.

Comme nous terminons le tutoriel Jobeet, nous allons avoir un autre regard sur le framework. Oubliez Jobeet pendant une heure, et rappelez vous de toutes les fonctionnalités que vous avez apprises au cours des trois dernières semaines.

Qu'est-ce que symfony ?

Le framework symfony est un ensemble cohérent mais découplé en sous-framework, qui forme un Framework MVC full-stack (Modèle, Vue, Contrôleur).

Avant de coder la tête la première, prenez le temps de lire l'histoire de symfony et la philosophie. Ensuite, vérifiez les prérequis du framework et utiliser le script `check_configuration.php` pour valider votre configuration.

Finalement, installer symfony. Après quelque temps, vous voulez aussi mettre à jour la dernière version du framework.

Le framework offre également des outils pour faciliter son déploiement.

Le Modèle

La partie du modèle de symfony peut être fait avec l'aide de l'ORM Doctrine. Selon la description de la base de données, il génère des classes pour les objets, les formulairess et les filtres. Doctrine génère également les instructions SQL utilisées pour créer les tables dans la base de données.

La configuration de base de données peut être fait avec une tâche ou en modifiant un fichier de configuration. A côté de sa configuration, il est également possible d'injecter des données initiales, grâce aux fichiers fixture. Vous pouvez même rendre ces fichiers dynamiques.

Les objets Doctrine peuvent aussi facilement être internationalisés.

La Vue

Par défaut, la couche de la vue de l'architecture MVC utilise de simples fichiers PHP comme Templates.

Les Templates peuvent utiliser les helpers pour les tâches récurrentes comme la création d'une URL ou un lien.

Un Template peut être décorée par un layout pour extraire l'entête et le pied des pages. Pour faire des vues encore plus réutilisables, vous pouvez définir des slots, des partials et des components.

Pour accélérer les choses, vous pouvez utiliser le sous-framework de cache pour mettre en cache une page entière, juste l'action, ou même juste des partials ou des components. Vous pouvez également supprimer le cache manuellement.

Le Contrôleur

La partie contrôleur est géré par les contrôleurs frontaux et les actions.

Les tâches peuvent être utilisées pour créer des modules simples, des modules CRUD ou même pour générer des modules admin complétement fonctionnels pour les classes de modèle.

Les modules admin vous permettent de construire une application entièrement fonctionnelle sans

rien coder.

Pour faire abstraction de la réalisation technique d'un site web, symfony utilise un sous-framework de routage qui génère des jolies URL. Pour faire l'implémentation encore plus facilement des services web, symfony prend en charge les formats. Vous pouvez également créer vos propres formats.

Une action peut être transmise à une autre ou redirigée.

Configuration

Le framework symfony permet facilement d'avoir différents paramètres de configuration pour différents environnements. Un environnement est un ensemble de paramètres qui permet des comportements différents sur le développement ou les serveurs de production. Vous pouvez également créer de nouveaux environnements.

Les fichiers de configuration de symfony peuvent être définis à différents niveaux et la plupart d'entre eux sont sensibles à l'environnement :

- `app.yml`
- `cache.yml`
- `databases.yml`
- `factories.yml`
- `generator.yml`
- `routing.yml`
- `schema.yml`
- `security.yml`
- `settings.yml`
- `view.yml`

Les fichiers de configuration pour la plupart utilisent le format YAML.

Au lieu d'utiliser la structure de répertoire par défaut et d'organiser vos fichier de l'application par couches, vous pouvez également les organiser par fonctionnalité et pouvoir les regrouper dans un plugin. Comme nous parlons de la structure de répertoire par défaut, vous pouvez aussi la personnaliser selon vos besoins.

Débogage

De la journalisation au web debug toolbar, en passant par les exceptions significatives, symfony dispose de nombreux outils utiles pour aider le développeur à déboguer les problèmes plus rapidement.

Principaux objets de symfony

Le framework symfony fournit assez peu d'objets du noyau qui abstraitent les besoins récurrents dans les projets web : la requête, la réponse, l'utilisateur, la journalisation, le routage, l'envoi de courrier et le gestionnaire du cache de la vue.

Ces objets du noyau sont gérées par l'objet `sfContext`, et ils sont configurés via lesfactories.

Le user gère l'authentification des utilisateurs, l'autorisation, les flashes, et les attributs pour être sérialisés dans la session.

Sécurité

Le framework symfony intègre des protections contre les attaques XSS et CSRF. Ces paramètres peuvent être configurés à partir de la ligne de commande, ou en modifiant un fichier de configuration.

Le framework de formulaire fournit également des fonctionnalités intégrées de sécurité.

Formulaires

Comme la gestion des formulaires est l'une des tâches les plus fastidieuses pour un développeur web, symfony fournit un sous-framework de formulaire. Le framework de formulaire est livré avec un lot de widgets et de validateurs. L'un des atouts du sous-framework de formulaire est que les Templates sont très facilement personnalisables.

Si vous utilisez Doctrine, le framework de formulaire permet également de générer des formulaires et des filtres basés depuis vos modèles.

Internationalisation et régionalisation

L'internationalisation et la régionalisation sont supportés par symfony, grâce à la norme ICU. La culture de l'utilisateur détermine la langue et le pays de l'utilisateur. Elle peut être définie par l'utilisateur lui-même, ou incorporée dans l'URL.

Tests

La bibliothèque lime, utilisée pour les **tests unitaires**, fournit un grand nombre de méthodes de test. Les objets Doctrine peuvent aussi être testés à partir d'une base de données dédiée et avec des jeux de test dédiés.

Les tests unitaires peuvent être exécutés un par un ou tous ensembles.

Les **tests fonctionnels** sont écrit avec la classe `sfFunctionalTest`, qui utilise un simulateur de navigateur et permet l'introspection des objets du noyau de symfony grâce aux testeurs. Les testeurs existent pour l'objet de la requête, l'objet de la réponse, l'objet utilisateur, l'objet du formulaire actuel, la couche du cache et les objets Doctrine.

Vous pouvez également utiliser les outils de débogage pour la réponse et les formulaires.

Comme pour les tests unitaires, les tests fonctionnels peuvent être exécutés un par un ou tous ensembles.

Vous pouvez également exécuter tous les tests ensembles.

Plugins

Le framework symfony fournit seulement la fondation de vos applications web et s'appuie sur des plugins pour ajouter plus de fonctionnalités. Dans ce tutoriel, nous avons parlé `sfGuardPlugin`, `sfFormExtraPlugin` et `sfTaskExtraPlugin`.

Un plugin doit être activé après l'installation.

Le plugin est le meilleur moyen de contribuer en retour au projet symfony.

Tâches

Le CLI de symfony fournit une foule de tâches, et les plus utiles ont été expliquées dans ce tutoriel :

- `app:routes`
- `cache:clear`
- `configure:database`
- `generate:project`
- `generate:app`
- `generate:module`
- `help`
- `i18n:extract`
- `list`
- `plugin:install`
- `plugin:publish-assets`
- `project:deploy`
- `doctrine:build --all`
- `doctrine:build --all --and-load`
- `doctrine:build --forms`
- `doctrine:build-model`
- `doctrine:build-sql`
- `doctrine:data-load`
- `doctrine:generate-admin`
- `doctrine:generate-module`
- `doctrine:insert-sql`
- `test:all`
- `test:coverage`
- `test:functional`
- `test:unit`

Vous pouvez également créer vos propres tâches.

À bientôt

L'apprentissage par la pratique

Le framework symfony, comme le fait n'importe quel morceau d'un logiciel, a une courbe d'apprentissage. Dans le processus d'apprentissage, la première étape est d'apprendre à partir d'exemples pratiques avec un livre comme celui-ci. La deuxième étape consiste à la **pratique**. Rien ne remplacera jamais la pratique.

C'est ce que vous pouvez commencer à faire aujourd'hui. Pensez au plus simple des projets web qui fournit toujours une certaine valeur : un gestionnaire de liste de todo, un simple blog, un convertisseur de devise ou de temps, quel qu'il soit ... Choisissez en un et commencez par appliquer la connaissance que vous avez aujourd'hui. Utilisez les messages d'aide pour apprendre les différentes options, consultez le code généré par symfony, utilisez un éditeur de texte qui supporte l'auto-complémentation PHP comme Eclipse et consultez le guide de référence pour parcourir toutes les configurations prévues par le framework.

Profitez de tous le matériel gratuit que vous avez à votre disposition pour en savoir plus sur symfony.

La communauté

Avant de vous quitter, je voudrais parler d'une dernière chose à propos de symfony. Le framework a beaucoup de fonctionnalités intéressantes et beaucoup de documentation gratuite. Mais, l'un des actifs les plus précieux d'un Open-Source est d'avoir sa communauté. Et symfony a l'une des communautés la plus étonnante et active. Si vous commencez à utiliser symfony pour vos projets, envisager de rejoindre la communauté symfony :

- Abonnez-vous à la mailing-list des utilisateurs
- Abonnez-vous au flux du blog officiel
- Abonnez-vous au flux planète de symfony
- Venez et chattez sur le canal IRC #symfony de freenode

« Jour 22 : Le déploiement

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.

Attribution-Share Alike 3.0 Unported License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.

c. **"Creative Commons Compatible License"** means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.

d. **"Distribute"** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

e. **"License Elements"** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

f. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

g. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that

transmits the broadcast.

h. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

i. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

j. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

k. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

i. **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

iii. **Voluntary License Schemes**. The Licensor waives the right to collect royalties, whether

individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

b. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection

appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

Creative Commons may be contacted at http://creativecommons.org/.

Questions & Feedback

If you find a typo or an error, please register and open a ticket.

If you need support or have a technical question, please post to the official user mailing-list.