les Cahiers du Programmeur

Symfony

Mieux développer en PHP avec Symfony 1.2 et Doctrine

Fabien **Potencier**
Hugo **Hamon**

EYROLLES

Programmez intelligent avec les Cahiers du Programmeur

Symfony

Reconnu dans le monde pour sa puissance et son élégance, Symfony est issu de plus de dix ans de savoir-faire. Le framework open source de Sensio fédère une très forte communauté de développeurs PHP professionnels. Il leur offre des outils et un environnement MVC pour créer des applications web robustes, maintenables et évolutives.

Au fil d'une démarche rigoureuse et d'un exemple concret d'application web 2.0, ce cahier décrit le bon usage des outils Symfony mis à la disposition du développeur : de l'architecture MVC et autres design patterns à l'abstraction de base de données et au mapping objet-relationnel avec Doctrine, en passant par les tests unitaires et fonctionnels, la gestion des URL, des formulaires ou du cache, l'internationalisation ou encore la génération des interfaces d'administration…

Adapté du tutoriel Jobeet mis à jour en français — Téléchargez le code source !
http://www.symfony-project.org/jobeet/

Sommaire

Fabien Potencier est ingénieur civil des Mines de Nancy et diplômé du mastère Entrepreneurs à HEC. Il a créé le framework Symfony dont il est le développeur principal. Co-fondateur de Sensio, il dirige Sensio Labs, agence spécialisée dans les technologies Open Source.

Diplômé d'une licence spécialisée en développement informatique, **Hugo Hamon** a rejoint Sensio Labs en tant que développeur web. Passionné par PHP, il a fondé le site Apprendre-PHP.com et promeut le langage en milieu professionnel en s'investissant dans l'AFUP et dans la communauté Symfony.

symfony doctrine

SENSIOLABS

Conception couverture : Nordcompo

les Cahiers du Programmeur

Symfony

Collection « Les cahiers du programmeur »

G. Ponçon et J. Pauli. – **Zend Framework**. N°12392, 2008, 460 pages.

L. Jayr. **Flex 3**. *Applications Internet riches*. N°12409, 2009, 226 pages.

P. Roques. – **UML 2. Modéliser une application web.** N°12389, 6e édition, 2008, 247 pages

A. Goncalves. – **Java EE 5.** N°12363, 2e édition, 2008, 370 pages

E. Puybaret. **– Swing.** N°12019, 2007, 500 pages

E. Puybaret. – **Java 1.4 et 5.0.** N°11916, 3e édition, 2006, 400 pages

J. Molière. – **J2EE**. N°11574, 2e édition, 2005, 220 pages

R. Fleury – **Java/XML.** N°11316, 2004, 218 pages

J. Protzenko, B. Picaud. – **XUL**. N°11675, 2005, 320 pages

S. Mariel. – **PHP 5**. N°11234, 2004, 290 pages

chez le même éditeur

C. Porteneuve. – **Bien développer pour le Web 2.0.** N°12391, 2e édition 2008, 600 pages.

E. Daspet, C. Pierre de Geyer. – **PHP 5 avancé.** N°12369, 5e édition, 2008, 844 pages

G. Ponçon. – **Best practices PHP 5.** *Les meilleures pratiques de développement en PHP*. N°11676, 2005, 470 pages

T. Ziadé. – **Programmation Python**. – N°12483, 2e édition, 2009, 530 pages

C. Pierre de Geyer, G. Ponçon. – **Mémento PHP 5 et SQL.** N°12457, 2e édition, 2009, 14 pages

J.-M. Defrance. **– Premières applications Web 2.0 avec Ajax et PHP.** N°12090, 2008, 450 pages

D. Seguy, P. Gamache. – **Sécurité PHP 5 et MySQL**. N°12114, 2007, 250 pages

A. Vannieuwenhuyze. **Programmation Flex 3**. N°12387, 2008, 430 pages

V. Messager-Rota. – **Gestion de projet.** *Vers les méthodes agiles*. N°12158, 2e édition, 2009, 252 pages

H. Bersini, I. Wellesz. – **L'orienté objet**. N°12084, 3e édition, 2007, 600 pages

P. Roques. – **UML 2 par la pratique**. N°12322, 6e édition, 368 pages

S. Bordage. – **Conduite de projet Web.** N°12325, 5e édition, 2008, 394 pages

J. Dubois, J.-P. Retaillé, T. Templier. **– Spring par la pratique**. *Java/J2EE, Spring, Hibernate, Struts, Ajax*. – N°11710, 2006, 518 pages

A. Boucher. – **Mémento Ergonomie web**. N°12386, 2008, 14 pages

A. Fernandez-Toro. – **Management de la sécurité de l'information**. *Implémentation ISO 27001*. N°12218, 2007, 256 pages

Collection « Accès libre »

Pour que l'informatique soit un outil, pas un ennemi !

Économie du logiciel libre. F. Elie. N°12463, 2009, 195 pages

Hackez votre Eee PC. *L'ultraportable efficace*. C. Guelff. N°12437, 2009, 306 pages

Joomla et Virtuemart – *Réussir sa boutique en ligne*. V. Isaksen, T. Tardif. – N°12381, 2008, 270 pages

Open ERP – *Pour une gestion d'entreprise efficace et intégrée*. F. Pinckaers, G. Gardiner. – N°12261, 2008, 276 pages

Réussir son site web avec XHTML et CSS. M. Nebra. – N°12307, 2e édition, 2008, 316 pages

Ergonomie web. *Pour des sites web efficaces*. A. Boucher. – N°12479, 2e édition, 2009, 456 pages

Gimp 2 efficace – *Dessin et retouche photo*. C. Gémy. – N°12152, 2e édition, 2008, 402 pages

OpenOffice.org 3 efficace. S. Gautier, G. Bignebat, C. Hardy, M. Pinquier. – N°12408, 2009, 408 pages avec CD-Rom.

Réussir un site web d'association… avec des outils libres. A.-L. et D. Quatravaux. – N°12000, 2e édition, 2007, 372 pages

Réussir un projet de site Web. N. Chu. – N°12400, 5e édition, 2008, 230 pages

Fabien **Potencier**
Hugo **Hamon**

les Cahiers du Programmeur

Symfony

Mieux développer en PHP avec Symfony 1.2 et Doctrine

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Remerciements à Franck Bodiot pour certaines illustrations d'ouverture de chapitre.

Avant-propos

Après plus de trois ans d'existence en tant que projet Open Source, Symfony est devenu l'un des frameworks incontournables de la scène PHP. Son adoption massive ne s'explique pas seulement par la richesse de ses fonctionnalités ; elle est aussi due à l'excellence de sa documentation – probablement l'une des meilleures pour un projet Open Source.

La sortie de la première version officielle de Symfony a été célébrée avec la publication en ligne du tutoriel *Askeet*, qui décrit la réalisation d'une application sous Symfony en 24 étapes prévues pour durer chacune une heure. Publié à Noël 2005, ce tutoriel devint un formidable outil de promotion du framework. Nombre de développeurs ont en effet appris à utiliser Symfony grâce à Askeet, et certaines sociétés l'utilisent encore comme support de formation.

Le temps passant, et avec l'arrivée de la version 1.2 de Symfony, il fut décidé de publier un nouveau tutoriel sur le même format qu'Askeet. Le tutoriel *Jobeet* fut ainsi publié jour après jour sur le blog officiel de Symfony, du 1er au 24 décembre 2008 ; vous lisez actuellement sa version éditée sous forme de livre papier.

COMMUNAUTÉ

Une étude de cas communautaire

Pour Askeet, il avait été demandé à la communauté des utilisateurs de Symfony de proposer une fonctionnalité à ajouter au site. L'initiative eut du succès et le choix se porta sur l'ajout d'un moteur de recherche. Le vœu de la communauté fut réalisé, et le chapitre consacré au moteur de recherche est d'ailleurs rapidement devenu l'un des plus populaires du tutoriel.

Dans le cas de Jobeet, l'hiver a été célébré le 21 décembre avec l'organisation d'un concours de design où chacun pouvait soumettre une charte graphique pour le site. Après un vote communautaire, la charte de l'agence américaine *centre{source}* fut choisie. C'est cette interface graphique qui sera intégrée tout au long de ce livre.

Découvrir l'étude de cas développée

Cet ouvrage décrit le développement d'un site web avec Symfony, depuis ses spécifications jusqu'à son déploiement en production, en 21 chapitres d'une heure environ. Au travers des besoins fonctionnels du site à développer, chaque chapitre sera l'occasion de présenter non seulement les fonctionnalités de Symfony mais également les bonnes pratiques du développement web.

L'application développée dans cet ouvrage aurait pu être un moteur de blog – exemple souvent choisi pour d'autres frameworks ou langages de programmation. Nous souhaitions cependant un projet plus riche et plus original, afin de démontrer qu'il est possible de développer facilement et rapidement des applications web professionnelles avec Symfony. C'est au chapitre 2 que vous en découvrirez les spécificités ; pour le moment, seul son nom de code est à mémoriser : *Jobeet*...

En quoi cet ouvrage est-il différent ?

On se souvient tous des débuts du langage PHP 4. C'était la belle époque du Web ! PHP a certainement été l'un des premiers langages de programmation dédié au Web et sûrement l'un des plus simples à maîtriser.

Mais les technologies web évoluant très vite, les développeurs ont besoin d'être en permanence à l'affût des dernières innovations et surtout des bonnes pratiques. La meilleure façon d'effectuer une veille technologique efficace est de lire des blogs d'experts, des tutoriels éprouvés et bien évidemment des ouvrages de qualité. Cependant, pour des langages aussi variés que le PHP, le Python, le Java, le Ruby, ou même le Perl, il est décevant de constater qu'un grand nombre de ces ouvrages présentent une lacune majeure... En effet, dès qu'il s'agit de montrer des exemples de code, ils laissent de côté des sujets primordiaux, et pallient le manque par des avertissements de ce genre :

- « Lors du développement d'un site, pensez aussi à la validation et la détection des erreurs » ;
- « Le lecteur veillera bien évidemment à ajouter la gestion de la sécurité » ;
- « L'écriture des tests est laissée à titre d'exercice au lecteur. »

Or chacune de ces questions – validation, sécurité, gestion des erreurs, tests – est primordiale dès qu'il s'agit d'écrire du code professionnel. Comment ne pas se sentir, en tant que lecteur, un peu abandonné ? Si ces contraintes – de surcroît les plus complexes à gérer pour un développeur – ne sont pas prises en compte, les exemples perdent de leur intérêt et de leur exemplarité !

Le livre que vous tenez entre les mains ne contient pas d'avertissement de ce type : une attention particulière est prêtée à l'écriture du code nécessaire pour gérer les erreurs et pour valider les données entrées par l'utilisateur. Du temps est également consacré à l'écriture de tests automatisés afin de valider les développements et les comportements attendus du système.

BONNE PRATIQUE **Réutilisez le code libre quand il est exemplaire !**

Le code que vous découvrirez dans ce livre peut servir de base à vos futurs développements ; n'hésitez surtout pas à en copier-coller des bouts pour vos propres besoins, voire à en récupérer des fonctionnalités complètes si vous le souhaitez.

Symfony fournit en standard des outils permettant au développeur de tenir compte de ces contraintes plus facilement et en étant parcimonieux en quantité de code. Une partie de cet ouvrage est consacrée à ces fonctionnalités car encore une fois, la validation des données, la gestion des erreurs, la sécurité et les tests automatisés sont ancrés au cœur même du framework – ce qui lui permet d'être employé y compris sur des projets de grande envergure.

Dans la philosophie de Symfony, les bonnes pratiques de développement ont donc part égale avec les nombreuses fonctionnalités du framework. Elles sont d'autant plus importantes que Symfony est utilisé pour le développement d'applications critiques en entreprise.

Organisation de l'ouvrage

Cet ouvrage est composé de vingt-et-un chapitres qui expliquent pas à pas la construction d'une application web professionnelle Open Source avec le framework Symfony. L'objectif de cette série de chapitres est de détailler une à une les fonctionnalités qui font le succès de Symfony, mais aussi et surtout de montrer ce qui fait de Symfony un outil professionnel, efficace et agréable à utiliser.

Le **chapitre 1** ouvre le bal avec l'installation et l'initialisation du projet Jobeet. Ces premières pages sont jalonnées en cinq parties majeures : le téléchargement et l'installation des librairies de Symfony, la génération de la structure de base du projet ainsi que celle de la première application, la configuration du serveur web et enfin l'installation d'un dépôt Subversion pour le contrôle du suivi du code source.

Le **chapitre 2** dresse le cahier des charges fonctionnelles de l'application développée au fil des chapitres. Les besoins fonctionnels majeurs de Jobeet y seront décrits un à un à l'aide de cas d'utilisation illustrés.

Le **chapitre 3** entame véritablement les hostilités en s'intéressant à la conception du modèle de la base de données, et à la construction automatique de cette dernière à partir de l'ORM Doctrine. L'intégralité du chapitre sera ponctuée par de nombreuses astuces techniques et bonnes pratiques de développement web. Ce chapitre s'achèvera enfin avec la génération du tout premier module fonctionnel de l'application à l'aide des tâches automatiques de Symfony.

Le **chapitre 4** aborde l'un des points clés du framework Symfony : l'implémentation du motif de conception Modèle Vue Contrôleur. Ces quelques pages expliqueront tous les avantages qu'apporte cette méthodologie éprouvée en termes d'organisation du code par rapport à une autre, et sera l'occasion de découvrir et de mettre en œuvre les couches de la Vue et du Contrôleur.

Le **chapitre 5** se consacre quant à lui à un autre sujet majeur de Symfony : le routage. Cet aspect du framework concerne la génération des URLs propres et la manière dont elles sont traitées en interne par Symfony. Ce chapitre sera donc l'occasion de présenter les différentes types de routes qu'il est possible de créer et de découvrir comment certaines d'entre elles sont capables d'interagir directement avec la base de données pour retrouver des objets qui leur sont liés.

Le **chapitre 6** est dédié à la manipulation de la couche du Modèle avec Symfony. Ce sera donc l'occasion de découvrir en détail comment le framework Symfony et l'ORM Doctrine permettent au développeur de manipuler une base de données en toute simplicité à l'aide d'objets plutôt que de requêtes SQL brutes. Ce chapitre met également l'accent sur une autre bonne pratique ancrée dans la philosophie du framework Symfony : le remaniement du code. Le but de cette partie du chapitre est de sensibiliser le lecteur à l'intérêt d'une constante remise en question de ses développements – lorsqu'il a la possibilité de l'améliorer et de le simplifier.

Le **chapitre 7** est une compilation de tous les sujets abordés précédemment puisqu'il y est question du modèle MVC, du routage et de la manipulation de la base de données par l'intermédiaire des objets. Toutefois, les pages de ce chapitre introduisent deux nouveaux concepts : la simplification du code de la Vue ainsi que la pagination des listes de résultats issus d'une base de données. De la même manière qu'au sixième chapitre, un remaniement régulier du code sera opéré afin de comprendre tous les bénéfices de cette bonne pratique de développement.

Le **chapitre 8** présente à son tour un sujet encore méconnu des développeurs professionnels mais particulièrement important pour garantir la qualité des développements : les tests unitaires. Ces quelques pages présentent tous les avantages de l'ajout de tests automatiques pour une application web, et expliquent de quelle manière ces derniers sont parfaitement intégrés au sein du framework Symfony via la librairie Open Source Lime.

Le **chapitre 9** fait immédiatement suite au précédent en se consacrant à un autre type de tests automatisés : les tests fonctionnels. L'objectif de ce chapitre est de présenter ce que sont véritablement les tests fonctionnels et ce qu'ils apportent comme garanties au cours du développement de l'application Jobeet. Symfony est en effet doté d'un sous-framework de tests fonctionnels puissant et simple à prendre en main, qui permet au développeur d'exécuter la simulation de l'expérience utilisateur dans son navigateur, puis d'analyser toutes les couches de l'application qui sont impliquées lors de ces scénarios.

Pour ne pas interrompre le lecteur dans sa lancée et sa soif d'apprentissage, le **chapitre 10** aborde l'importante notion de gestion des formulaires. Les formulaires constituent la principale partie dynamique d'une application web puisqu'elle permet à l'utilisateur final d'interagir avec le système. Bien que les formulaires soient faciles à mettre en place, leur gestion n'en demeure pas moins très complexe puisqu'elle implique des notions de validation de la saisie des utilisateurs, et donc de sécurité. Heureusement, Symfony intègre un sous-framework destiné aux formulaires capable de simplifier et d'automatiser leur gestion en toute sécurité.

Le **chapitre 11** agrège les connaissances acquises aux chapitres 9 et 10 en expliquant de quelle manière il est possible de tester fonctionnellement des formulaires avec Symfony. Par la même occasion, ce sera le moment idéal pour écrire une première tâche automatique de maintenance, exécutable en ligne de commande ou dans une tâche planifiée du serveur.

Le **chapitre 12** est l'un des plus importants de cet ouvrage puisqu'il fait le tour complet d'une des fonctionnalités les plus appréciées des développeurs Symfony : le générateur d'interface d'administration. En quelques minutes seulement, cet outil permettra de bâtir un espace complet et sécurisé de gestion des catégories et des offres d'emploi de Jobeet.

L'utilisateur est l'acteur principal dans une application puisque c'est lui qui interagit avec le serveur et qui récupère ce que ce dernier lui renvoie en retour. Par conséquent, le **chapitre 13** se dédie entièrement à lui et montre, entre autres, comment sauvegarder des informations persistantes dans la session de l'utilisateur, ou encore comment lui restreindre l'accès à certaines pages s'il n'est pas authentifié ou s'il ne dispose pas des droits d'accès nécessaires et suffisants. D'autre part, une série de remaniements du code sera réalisée pour simplifier davantage le code et le rendre testable.

Le **chapitre 14** s'intéresse à une puissante fonctionnalité du sous-framework de routage : le support des formats de sortie et l'architecture RESTful. À cette occasion, un module complet de génération de flux de syndication RSS/ATOM est développé en guise d'exemple afin de montrer avec quelle simplicité Symfony est capable de gérer nativement différents formats de sortie standards.

Le **chapitre 15** approfondit les connaissances sur le framework de routage et les formats de sortie en développant une API de services web destinés aux webmasters, qui leur permet d'interroger Jobeet afin d'en récupérer des résultats dans un format de sortie XML, JSON ou YAML. L'objectif est avant tout de montrer avec quelle aisance Symfony facilite la création de services web innovants grâce à son architecture RESTful.

Toute application dynamique qui se respecte comprend spontanément un moteur de recherche, et c'est exactement l'objectif du **chapitre 16**. En seulement quelques minutes, l'application Jobeet bénéficiera d'un moteur de recherche fonctionnel et testé, reposant sur le composant `Zend_Search_Lucene` du framework Open Source de la société Zend. C'est l'un des nombreux avantages de Symfony que de pouvoir accueillir simplement des composants tiers comme ceux du framework Zend.

Le **chapitre 17** améliore l'expérience utilisateur du moteur de recherche créé au chapitre précédent, en intégrant des composants JavaScript et Ajax non intrusifs, développés au moyen de l'excellente librairie jQuery. Grâce à ces codes JavaScript, l'utilisateur final de Jobeet bénéficiera d'un moteur de recherche dynamique qui filtre et rafraîchit la liste de résultats en temps réel à chaque fois qu'il saisira de nouveaux caractères dans le champ de recherche.

Le **chapitre 18** aborde un nouveau point commun aux applications web professionnelles : l'internationalisation et la localisation. Grâce à Symfony, l'application Jobeet se dotera d'une interface multilingue dont les contenus traduits seront gérés à la fois par Doctrine pour les informations dynamiques des catégories, et par le biais de catalogues XLIFF standards.

Le **chapitre 19** se consacre à la notion de plug-ins dans Symfony. Les plug-ins sont des composants réutilisables à travers les différents projets, et qui constituent également un moyen d'organisation du code différent de la structure par défaut proposée par Symfony. Par conséquent, les pages de ce chapitre expliquent pas à pas tout le processus de transformation de l'application Jobeet en plug-in complètement indépendant et réutilisable.

Le **chapitre 20** de cet ouvrage se consacre au puissant sous-framework de mise en cache des pages HTML afin de rendre l'application encore plus performante lorsqu'elle sera déployée en production au dernier chapitre. Ce chapitre est aussi l'occasion de découvrir de quelle manière de nouveaux environnements d'exécution peuvent être ajoutés au projet, puis soumis à des tests automatisés.

Enfin, le **chapitre 21** clôture cette étude de cas par la préparation de l'application à la dernière étape décisive d'un projet web : le déploiement en production. Les pages de ce chapitre introduisent tous les concepts de configuration du serveur web de production ainsi que les outils d'automatisation des déploiements tels que rsync.

Pour conclure, trois parties d'**annexes** sont disponibles à la fin de cet ouvrage pour en savoir plus sur la syntaxe du format YAML et sur les directives de paramétrage de deux fichiers de configuration de Symfony présents dans chaque application développée.

Remerciements

Écrire un livre est une activité aussi excitante qu'épuisante. Pour un ouvrage technique, c'est d'autant plus intense qu'on cherche, heure après heure, à comprendre comment faire passer son message, comment expliquer les différents concepts, et comment fournir des exemples à la fois simples, pertinents et réutilisables.

Écrire un livre est une tâche tout simplement impossible à réaliser sans l'aide de certaines personnes qui vous entourent et vous soutiennent tout au long de ce processus.

Le plus grand soutien que l'on peut obtenir vient bien sûr de sa propre famille, et je sais que j'ai l'une des familles les plus compréhensives et encourageantes qui soient. En tant qu'entrepreneur, je passe déjà la plupart de mon temps au bureau, et en tant que principal développeur de Symfony, je passe une grande partie de mon temps libre à concevoir la prochaine version du framework. À cela s'ajoute ma décision d'écrire un nouveau livre. Mais sans les encouragements constants de ma femme Hélène et de mes deux merveilleux fils, Thomas et Lucas, ce livre n'aurait jamais été écrit en si peu de temps et n'aurait jamais pu voir le jour si rapidement.

Cet ouvrage n'aurait pu être réalisé sans le soutien d'autres personnes que je tiens particulièrement à remercier. En tant que président-directeur général de Sensio, j'ai de nombreuses responsabilités, et grâce à l'appui de toute l'équipe de Sensio, j'ai pu mener à terme ce projet. Mes principaux remerciements vont tout droit à Grégory Pascal, mon partenaire depuis dix ans, qui était au début particulièrement sceptique quant à l'idée d'entreprendre avec le « business model » de l'Open Source ; il m'en remercie énormément aujourd'hui.

Je souhaite aussi remercier Laurent Vaquette, mon aide de camp, qui n'a cessé de me simplifier la vie chaque jour, et d'accepter de m'accompagner de temps en temps pour manger un döner kebab.

Je remercie également Jonathan Wage, le développeur principal du projet Doctrine, qui a pris part à l'écriture de cet ouvrage. Grâce à ses nombreux efforts, la communauté Symfony bénéficie aujourd'hui de l'ORM Doctrine en natif dans Symfony ainsi que d'une véritable source de documentation par l'intermédiaire de cet ouvrage.

Enfin, Hugo Hamon, qui a été le principal artisan de cette transformation de la version originale anglaise, et à qui il me semble juste de laisser une place de co-auteur à mes côtés, sur ce premier ouvrage en français.

Fabien Potencier

Je tiens avant tout à remercier ma famille, mes amis et mes proches qui m'ont soutenu et encouragé de près comme de loin dans cette aventure à la fois passionnante, excitante et terriblement fatigante. J'en profite d'ailleurs pour dédicacer cet ouvrage à mes deux frères Hadrien et Léo.

J'adresse également mes remerciements et ma reconnaissance à toute l'équipe de Sensio, et particulièrement à Grégory Pascal et Fabien Potencier qui ont su me faire confiance dès mon arrivée dans leur entreprise, et me faire découvrir le plaisir de travailler sur des projets web passionnants.

Hugo Hamon

Nous n'oublions pas bien sûr d'adresser nos remerciements aux équipes des éditions Eyrolles qui nous ont permis de mener ce livre à son terme, et tout particulièrement à Muriel Shan Sei Fan pour avoir piloté ce projet dans les meilleures conditions et dans la bonne humeur. Nous remercions également Romain Pouclet qui n'a cessé de produire un travail remarquable de relecture technique et d'indexation du contenu.

Et enfin, nous vous remercions, vous lecteurs, d'avoir acheté cet ouvrage. Nous espérons sincèrement que vous apprécierez les lignes que vous vous apprêtez à lire, et bien sûr que vous trouverez votre place parmi l'incroyable communauté des développeurs Symfony.

Fabien Potencier et Hugo Hamon

Table des matières

chapitre 1

symfony

Démarrage du projet

Un projet web nécessite dès le démarrage une plate-forme de développement complète dans la mesure où de nombreuses technologies interviennent et cohabitent ensemble.
Ce chapitre introduit les notions élémentaires de projet Symfony, d'environnements web, de configuration de serveur virtuel mais aussi de gestion du code source au moyen d'outils comme Subversion.

MOTS-CLÉS :

- Symfony, Apache, Subversion
- Vulnérabilités XSS et CSRF
- Bonnes pratiques de développement

Comme pour tout projet web, il est évident de ne pas se lancer tête baissée dans le développement de l'application, c'est pourquoi aucune ligne de code PHP ne sera dévoilée avant le troisième chapitre de cet ouvrage. Néanmoins, ce chapitre révèlera combien il est bénéfique et utile de profiter d'un framework comme Symfony seulement en créant un nouveau projet.

L'objectif de ce chapitre est de mettre en place l'environnement de travail et d'afficher dans le navigateur une page générée par défaut par Symfony. Par conséquent, il sera question de l'installation du framework Symfony, puis de l'initialisation de la première application mais aussi de la configuration adéquate du serveur web local. Pour finir, une section détaillera pas à pas comment installer rapidement un dépôt Subversion capable de gérer le contrôle du suivi du code source du projet.

Installer et configurer les bases du projet

Les prérequis techniques pour démarrer

Tout d'abord, il faut s'assurer que l'ordinateur de travail possède un environnement de développement web complet composé d'un serveur web (Apache par exemple), d'une base de données (MySQL, PostgreSQL, ou SQLite) et bien évidemment de PHP en version 5.2.4 ou supérieure.

Tout au long du livre, la ligne de commande permettra de réaliser de très nombreuses tâches. Elle sera particulièrement facile à appréhender sur un environnement de type Unix. Pour les utilisateurs sous environnement Windows, pas de panique, puisqu'il s'agit juste de taper quelques commandes après avoir démarré l'utilitaire `cmd` (*Démarrer > Exécuter > cmd*).

Ce livre étant une introduction au framework Symfony, les notions relatives à PHP 5 et à la programmation orientée objet sont considérées comme acquises.

Installer les librairies du framework Symfony

La première étape technique de ce projet démarre avec l'installation des librairies du framework Symfony. Pour commencer, le dossier dans lequel figureront tous les fichiers du projet doit être créé. Les utilisateurs de Windows et d'Unix disposent tous de la même commande `mkdir` pour y parvenir.

ASTUCE Installer une plate-forme de développement pour Windows

Des outils comme WAMP Server 2 (www.wampserver.com) sous Windows permettent d'installer en quelques clics un environnement Apache, PHP et MySQL complet utilisant les dernières versions de PHP. Ils permettent ainsi de démarrer immédiatement le développement de projets PHP sans avoir à se préoccuper de l'installation des différents serveurs.

REMARQUE Bénéficier des outils d'Unix sous Windows

Si vous souhaitez reproduire un environnement Unix sous Windows, et avoir la possibilité d'utiliser des utilitaires comme `tar`, `gzip` ou `grep`, vous pouvez installer `Cygwin` (http://cygwin.com). La documentation officielle est un peu restreinte, mais vous trouverez un très bon guide d'installation à l'adresse http://www.soe.ucsc.edu/~you/notes/cygwin-install.html. Si vous êtes un peu plus aventurier dans l'âme, vous pouvez même essayer Windows Services for Unix à l'adresse http://technet.microsoft.com/en-gb/interopmigration/bb380242.aspx.

Création du dossier du projet en environnement Unix

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

Création du dossier du projet en environnement Windows

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

Une fois le répertoire du projet créé, le répertoire `lib/vendor/` contenant les librairies de Symfony doit à son tour être construit dans le répertoire du projet.

Création du répertoire lib/vendor/ du projet

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

La page d'installation de Symfony (http://www.symfony-project.org/installation) sur le site officiel du projet liste et compare les différentes versions disponibles du framework. Ce livre a été écrit pour fonctionner avec la toute dernière version 1.2 de Symfony. À l'heure où sont écrites ces lignes, la dernière version de Symfony disponible est la 1.2.5.

La section *Source Download* de cette page propose un lien permettant de télécharger une archive des fichiers source de Symfony au format `.tgz` ou `.zip`. Cette archive doit être téléchargée dans le répertoire `lib/vendor/` qui vient d'être créé, puis décompressée dans ce même répertoire.

Installation des fichiers sources de Symfony dans le répertoire lib/vendor/

```
$ cd lib/vendor
$ tar zxpf symfony-1.2.5.tgz
$ mv symfony-1.2.5 symfony
$ rm symfony-1.2.5.tgz
```

Sous Windows, il est plus facile d'utiliser l'explorateur de fichiers pour décompresser l'archive au format ZIP. Après avoir renommé le répertoire en `symfony`, la structure du projet devrait ressembler à celle-ci : `c:\development\sfprojects\jobeet\lib\vendor\symfony`.

La configuration par défaut de PHP variant énormément d'une installation à une autre, il convient de s'assurer que la configuration du serveur correspond aux prérequis minimaux de Symfony. Pour ce faire, le script de vérification fourni avec Symfony doit être exécuté depuis la ligne de commande.

ASTUCE Éviter les chemins contenant des espaces

Pour des raisons de simplicité et d'efficacité dans la ligne de commande Windows, il est vivement recommandé aux utilisateurs d'environnements Microsoft d'installer le projet et d'exécuter les commandes Symfony dans un chemin qui ne contient aucun espace. Par conséquent, les répertoires `Documents and Settings` ou encore `My Documents` sont à proscrire.

Vérification de la configuration du serveur

```
$ cd ../..
$ php lib/vendor/symfony/data/bin/check_configuration.php
```

En cas de problème, le script rapportera toutes les informations nécessaires pour corriger l'erreur. Il faut également exécuter ce script depuis le navigateur web puisque la configuration de PHP peut être différente en fonction des deux environnements. Il suffit pour cela de copier le script quelque part sous la racine web et d'accéder à ce fichier avec le navigateur. Il ne faut pas oublier ensuite de le supprimer une fois la vérification terminée.

```
$ rm web/check_configuration.php
```

```
********************************
*                              *
*  symfony requirements check  *
*                              *
********************************

php.ini used by PHP: /apache2/php/etc/php.ini


** Mandatory requirements **

  OK        requires PHP >= 5.2.4
  OK        php.ini: requires zend.ze1_compatibility_mode set to off

** Optional checks **

  OK        PDO is installed
  OK        PDO has some drivers installed: sqlite2, sqlite, mysql
  OK        PHP-XML module installed
[[WARNING]] XSL module installed
            *** Install the XSL module (recommended for Propel) ***
  OK        can use token_get_all()
  OK        can use mb_strlen()
  OK        can use iconv()
  OK        can use utf8_decode()
  OK        has a PHP accelerator
  OK        php.ini: short_open_tag set to off
  OK        php.ini: magic_quotes_gpc set to off
  OK        php.ini: register_globals set to off
  OK        php.ini: session.auto_start set to off
```

Figure 1–1
Résultat du contrôle de la configuration du serveur

Une fois la configuration du serveur validée, il ne reste plus qu'à vérifier que Symfony fonctionne correctement en ligne de commande en utilisant le script `symfony` pour afficher la version du framework. Attention, cet exécutable prend un `V` majuscule en paramètre.

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

Sous Windows :

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

L'exécution du script `symfony` sans paramètre donne l'ensemble des possibilités offertes par cet utilitaire. Le résultat obtenu dresse la liste des tâches automatisées et des options offertes par le framework pour accélérer les développements.

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

Sous Windows :

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

Cet utilitaire est le meilleur ami du développeur Symfony. Il fournit de nombreux outils permettant d'améliorer la productivité des activités récurrentes comme la suppression du cache, la génération de code, etc.

Installation du projet

Dans Symfony, les *applications* partagent le même modèle de données et sont regroupées en *projet*. Le projet *Jobeet* accueillera deux applications au total. La première, nommée *frontend*, est l'application qui sera visible par tous les utilisateurs, tandis que la seconde, intitulée *backend*, est celle qui permettra aux administrateurs de gérer le site.

Générer la structure de base du projet

Pour l'instant, seules les librairies du framework Symfony sont installées dans le répertoire du projet, mais ce dernier ne dispose pas encore des fichiers et répertoires qui lui sont propres. Il faut donc demander à Symfony de bâtir toute la structure de base du projet comprenant de nombreux fichiers et répertoires qui seront tous étudiés au fur et à mesure des chapitres de cet ouvrage. La commande `generate:project` de l'exécutable `symfony` permet de créer ladite structure du projet.

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

Sous Windows :

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

La tâche `generate:project` génère la structure par défaut des répertoires et crée les fichiers nécessaires à un projet Symfony. Le tableau ci-dessous dresse la liste des différents répertoires créés.

Tableau 1–1 Liste des répertoires par défaut d'un projet Symfony

Répertoire	Description
`apps/`	Contient toutes les applications du projet
`cache/`	Contient les fichiers mis en cache
`config/`	Contient les fichiers de configuration globaux du projet
`lib/`	Contient les librairies et classes du projet
`log/`	Contient les fichiers de logs du framework
`plugins/`	Contient les plug-ins installés
`Test/`	Contient les scripts de tests unitaires et fonctionnels
`web/`	Racine web du projet, c'est-à-dire tout ce qui est accessible depuis un navigateur web (voir ci-dessous)

La tâche `generate:project` a également créé un raccourci `symfony` à la racine du projet `Jobeet` pour faciliter l'écriture de la commande lorsqu'une tâche doit être exécutée. À partir de maintenant, au lieu d'utiliser le chemin complet pour exécuter la commande `symfony`, il suffira d'utiliser le raccourci `symfony`.

Générer la structure de base de la première application frontend

À présent, l'objectif est de générer la structure de base de la première application `frontend` du projet. Celle-ci sera présente dans le répertoire `apps/` généré juste avant. Une fois de plus, il convient de faire appel à l'exécutable `symfony` afin d'automatiser la génération des répertoires et des fichiers propres à chaque application.

```
$ php symfony generate:app --escaping-strategy=on
  ➥ --csrf-secret="Unique$ecret" frontend
```

Une fois de plus, la tâche `generate:app` crée la structure par défaut des répertoires de l'application dans le dossier `apps/frontend/`.

Tableau 1–2 Liste des répertoires par défaut d'une application Symfony

Répertoire	Description
`config/`	Contient les fichiers de configuration de l'application
`lib/`	Contient les librairies et classes de l'application
`modules/`	Contient le code de l'application (MVC)
`templates/`	Contient les templates principaux

REMARQUE Pourquoi Symfony génère-t-il autant de fichiers ?

Un des bénéfices d'utiliser un framework hiérarchisé est de standardiser les développements. Grâce à la structure par défaut des fichiers et des répertoires de Symfony, n'importe quel développeur connaissant Symfony pourra reprendre un projet Symfony. En quelques minutes, il sera à même de naviguer dans le code, de corriger les bogues, ou encore d'ajouter de nouvelles fonctionnalités.

ASTUCE Utiliser l'exécutable à la racine du projet

Le fichier `symfony` est exécutable, les utilisateurs d'Unix peuvent remplacer chaque occurrence `php symfony` par `./symfony` dès maintenant. Pour Windows, il faut d'abord copier le fichier `symfony.bat` dans le projet et utiliser `symfony` à la place de `php symfony`.

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

REMARQUE Exécution des commandes Symfony

Toutes les commandes Symfony doivent être exécutées depuis le répertoire racine du projet, sauf si le contraire est clairement indiqué.

Lorsque la tâche `generate:app` a été appelée, deux options dédiées à la sécurité lui ont été passées en paramètres. Ces deux options permettent d'automatiser la configuration de l'application à sa génération.

- `--escaping-strategy` : cette option active les échappements pour prévenir des attaques `XSS`.
- `--csrf-secret` : cette option active la génération des jetons de session des formulaires pour prévenir des attaques `CSRF`.

En passant ces deux options à la tâche, les futurs développements qui seront réalisés tout au long de cet ouvrage seront désormais protégés des vulnérabilités les plus courantes sur le web. Le framework Symfony se charge automatiquement de prendre les mesures de sécurité à la place du développeur pour lui éviter de se soucier de ces problématiques récurrentes.

CULTURE WEB En savoir plus sur les attaques XSS et CSRF

Les attaques XSS (*Cross Site Scripting*), et les attaques CSRF (*Cross Site Request Forgeries* ou *Sea Surf*), sont à la fois les plus répandues sur le web mais aussi les plus dangereuses. Par conséquent, il est important de bien les connaître pour savoir s'en prémunir efficacement. L'encyclopédie en ligne Wikipédia consacre une page dédiée à chacune d'elles aux adresses suivantes :
http://en.wikipedia.org/wiki/Cross-site_scripting
http://en.wikipedia.org/wiki/Cross-Site_Request_Forgery

Configuration du chemin vers les librairies de Symfony

La commande `symfony –V` permet de connaître la version du framework installée pour le projet, mais elle donne également le chemin absolu vers le répertoire des librairies de Symfony qui se trouve aussi dans le fichier de configuration `config/ProjectConfiguration.class.php`.

```
require_once '/Users/fabien/work/symfony/dev/1.2/lib/autoload/
sfCoreAutoload.class.php';
```

Le problème avec ce chemin absolu autogénéré est qu'il n'est pas portable puisqu'il correspond exclusivement à la configuration de la machine courante. Par conséquent, il convient de le changer au profit d'un chemin relatif, ce qui assurera le portage de tout le projet d'une machine à une autre sans avoir à modifier quoi que ce soit pour que tout fonctionne.

```
require_once dirname(__FILE__).'/../lib/vendor/symfony/lib/
autoload/sfCoreAutoload.class.php';
```

Découvrir les environnements émulés par Symfony

Le répertoire `web/` du projet contient deux fichiers créés automatiquement par Symfony à la génération de l'application `frontend` : `index.php` et `frontend_dev.php`. Ces deux fichiers sont appelés *contrôleurs frontaux* ou *front controllers* en anglais. Les deux termes seront employés dans cet ouvrage. Ces deux fichiers ont pour objectif de traiter toutes les requêtes HTTP qui les traversent et qui sont à destination de l'application. La question qui se pose alors est la suivante : Pourquoi avoir deux contrôleurs frontaux alors qu'une seule application a été générée ?

Quels sont les principaux environnements en développement web ?

Les deux fichiers pointent vers la même application à la différence qu'ils prennent chacun en compte un environnement différent. Lorsque l'on développe une application, à l'exception de ceux qui développent directement sur le serveur de production, il est nécessaire d'avoir plusieurs *environnements d'exécution cloisonnés* :

- *l'environnement de développement* est celui qui est utilisé par les développeurs quand ils travaillent sur l'application pour lui ajouter de nouvelles fonctionnalités ou corriger des bogues ;
- *l'environnement de test* sert quant à lui à soumettre l'application à des séries de tests automatisés pour vérifier qu'elle se comporte bien ;
- *l'environnement de recette* est celui qu'utilise le client pour tester l'application et rapporter les bogues et fonctionnalités manquantes aux chefs de projet et développeurs ;
- *l'environnement de production* est l'environnement sur lequel les utilisateurs finaux agissent.

Spécificités de l'environnement de développement

Qu'est-ce qui rend un environnement unique ? Dans l'environnement de développement par exemple, l'application a besoin d'enregistrer tous les détails de chaque requête afin de faciliter le débogage, tandis que le système de cache des pages est désactivé étant donné que les changements doivent être visibles immédiatement.

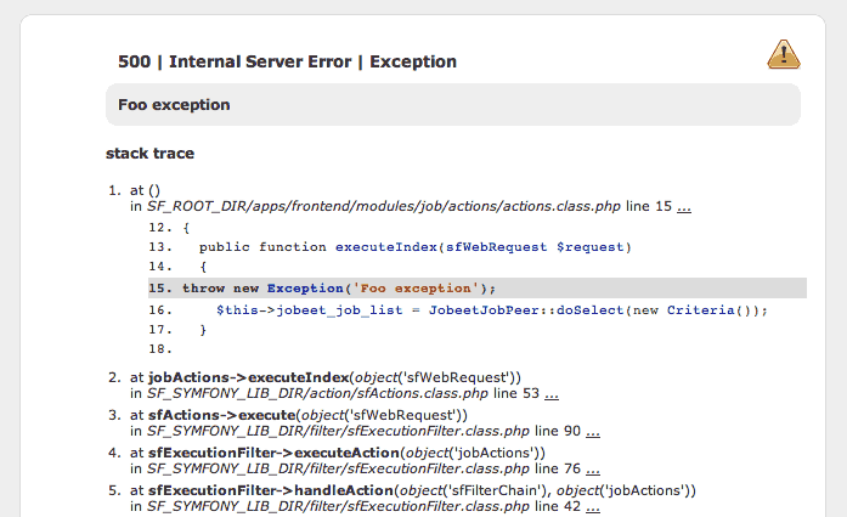


Figure 1–2
Affichage des informations de débogage en environnement de développement

Cet environnement est donc optimisé pour les besoins du développeur puisqu'il lui rapporte toutes les informations techniques dont il a besoin

pour travailler dans de bonnes conditions. Le meilleur exemple est bien sûr lorsqu'une exception PHP survient. Pour aider le développeur à déboguer le problème rapidement, le framework Symfony lui affiche dans le navigateur le message d'erreur avec toutes les informations qu'il dispose concernant la requête exécutée. La capture d'écran précédente en témoigne.

Spécificités de l'environnement de production

Sur l'environnement de production, la différence provient du fait que le cache des pages doit bien sûr être activé, et que l'application est configurée de telle sorte qu'elle affiche des messages d'erreur personnalisés aux utilisateurs finaux à la place des exceptions brutes. En d'autres termes, l'environnement de production doit être optimisé pour répondre aux problématiques de performance et favoriser l'expérience utilisateur. La capture d'écran ci-dessous donne le résultat de la même requête, exécutée précédemment en environnement de développement, sur l'environnement de production.

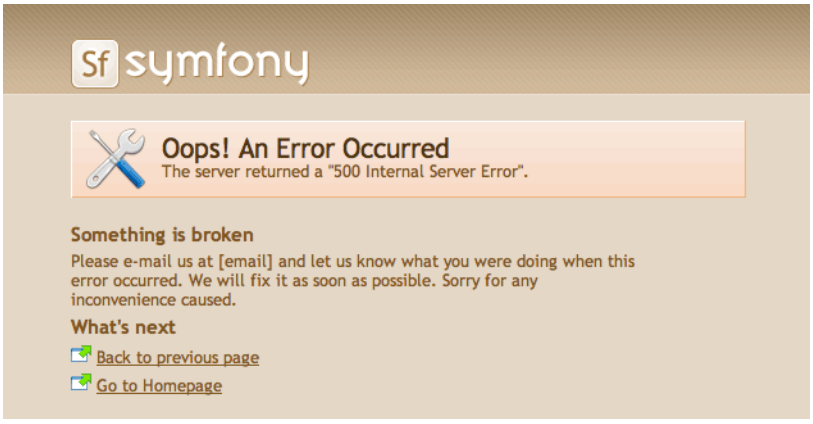


Figure 1–3
Affichage de la page d'erreur par défaut de Symfony en environnement de production

Un environnement Symfony est un ensemble unique de paramètres de configuration. Le framework Symfony est livré par défaut avec trois d'entre eux : `dev`, `test`, et `prod`. Au cours du chapitre 20, il sera présenté comment créer de nouveaux environnements tel que celui de la recette. Si l'on ouvre les différents fichiers des contrôleurs frontaux pour les comparer, on constate que leur contenu est strictement identique, à l'exception du paramètre de configuration dédié à l'environnement.

Astuce Créer de nouveaux environnements

Déclarer un nouvel environnement Symfony est aussi simple que de créer un nouveau contrôleur frontal. Plusieurs chapitres et annexes de cet ouvrage présentent comment modifier la configuration pour un environnement donné.

Contenu du contrôleur frontal web/index.php

```
<?php

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

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

Configurer le serveur web

Les sections qui suivent s'intéressent à la configuration du serveur web afin que celle-ci convienne parfaitement aux besoins d'un projet Symfony en termes de sécurité et de bonnes pratiques. Deux méthodes de configuration du serveur sont présentées. La première explique ce qu'il ne faut absolument pas faire, tandis que la seconde montre la bonne manière de procéder.

Méthode 1 : configuration dangereuse à ne pas reproduire

Dans la section précédente, un répertoire complet a été créé pour héberger l'ensemble du projet Jobeet. Si ce dernier a été construit quelque part sous la racine web du serveur, alors il est désormais accessible entièrement depuis un navigateur web.

Bien sûr, comme il n'y a aucune configuration et que c'est très facile à mettre en œuvre, cela signifie aussi que ce n'est pas la meilleure manière de procéder… Pour s'en convaincre, il suffit d'essayer d'accéder au fichier `config/databases.yml` depuis le navigateur pour comprendre les conséquences qui peuvent être provoquées avec ce type d'attitude paresseuse. En effet, si un utilisateur malintentionné découvre que le site web est développé avec Symfony, il aura alors accès à de nombreux fichiers de configuration sensibles en lecture…

Il est donc important de garder à l'esprit de ne jamais utiliser ce type de configuration sur un serveur de production, et ainsi de préférer l'étape de configuration décrite à la section suivante. En effet, cette dernière présente pas à pas comment configurer proprement le serveur web pour un projet Symfony.

Méthode 2 : configuration sûre et recommandée

Une bonne pratique de développement web consiste à placer sous la racine web uniquement les fichiers qui ont véritablement besoin d'être atteints depuis un navigateur : les feuilles de styles en cascade (CSS), les scripts JavaScript, les animations Flash ou encore les images. Bien sûr, par défaut, ces fichiers sont stockés sous le répertoire `web/` du projet Symfony.

Ce répertoire contient d'autres sous-dossiers dédiés aux ressources web (`css/` et `images/`) ainsi que les deux contrôleurs frontaux. Ces derniers sont les seuls fichiers PHP qui ont vocation à se trouver sous la racine web du serveur. Tous les autres fichiers PHP peuvent être cachés du navigateur, ce qui est plus que conseillé d'un point de vue sécurité.

Création d'un nouveau serveur virtuel pour Jobeet

À présent, il est temps de changer la configuration par défaut du serveur Apache pour rendre le nouveau projet accessible sur Internet. Pour cela, il suffit de localiser et d'ouvrir le fichier de configuration `http.conf` et d'y ajouter à la fin les paramètres de configuration suivants.

Définition de la configuration du serveur virtuel de Jobeet

```
# Be sure to only have this line once in your configuration
NameVirtualHost 127.0.0.1:8080

# This is the configuration for Jobeet
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>
```

Cette configuration indique au serveur Apache qu'il faut écouter le port `8080` de la machine. Par conséquent, le site Internet de Jobeet sera accessible à l'URL suivante : http://localhost:8080/. Le port peut être modifié par un nombre strictement supérieur à 1 024 étant donné qu'il ne requiert pas de droits d'administrateur.

CONFIGURATION

La directive de configuration Alias

L'alias `/sf` autorise l'accès aux images et fichiers JavaScript dont ont besoin pour s'afficher la page par défaut de Symfony et la barre de débogage. Sur les environnements Windows, la valeur de la directive Alias doit être remplacée par quelque chose du genre :

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

Et `/home/sfprojects/jobeet/web` devrait être remplacé au profit de :

```
c:\development\sfprojects\jobeet\web
```

CONFIGURATION **Configurer un nom de domaine pour l'application**

Pour les administrateurs de la machine, il est préférable de configurer des serveurs virtuels plutôt que d'ouvrir un nouveau port à chaque fois qu'un nouveau projet démarre. Au lieu de choisir un port et d'ajouter un écouteur supplémentaire, il vaut mieux trouver un nom de domaine et l'ajouter à la directive de configuration `ServerName`.

```
# This is the configuration for Jobeet
<VirtualHost 127.0.0.1:80>
  ServerName jobeet.localhost
  <!-- same configuration as before -->
</VirtualHost>
```

Le nom de domaine `jobeet.localhost` utilisé dans la configuration d'Apache doit être déclaré localement. Pour les environnements Linux, cela se passe dans le fichier `/ect/hosts`, tandis que pour Windows XP, ce fichier se trouve dans le répertoire `C:\WINDOWS\system32\drivers\etc\`.

La configuration du nom de domaine consiste à ajouter cette ligne supplémentaire au fichier `hosts`.

```
127.0.0.1         jobeet.localhost
```

Tester la nouvelle configuration d'Apache

Pour tester cette nouvelle configuration, le serveur Apache doit d'abord être redémarré afin de recharger les nouveaux paramètres de configuration. Il ne reste alors plus qu'à ouvrir le navigateur web et vérifier que le contrôleur frontal `index.php` est bien accessible sur le web.

Pour ce faire, il suffit d'appeler l'une des deux URLs suivantes en fonction de la configuration choisie précédemment pour Apache : http://localhost:8080/index.php/ ou http://jobeet.localhost/index.php/. La page d'accueil par défaut devrait apparaître à l'écran confirmant deux choses : d'une part que le serveur virtuel de Jobeet fonctionne, et d'autre part que l'alias `/sf` est bien configuré puisque le navigateur est capable de récupérer les ressources web de la page.

Il est également possible d'accéder à l'application en environnement de développement en utilisant l'URL suivante : http://jobeet.localhost/frontend_dev.php/. La barre de débogage de Symfony devrait s'afficher dans l'angle supérieur droit de la fenêtre du navigateur, incluant avec elle de petits icônes qui prouvent que l'alias `/sf` est convenablement configuré.

La barre de débogage de Symfony est présente sur chaque page en environnement de développement et donne au développeur l'accès à de nombreuses informations en cliquant sur les différents onglets. Parmi elles figurent la configuration courante de l'application, les traces de logs enregistrés pour la requête exécutée, les requêtes SQL exécutées sur la base de données, le temps de génération de la page ainsi que la quantité de mémoire utilisée.

ASTUCE **Profiter de la réécriture d'URL d'Apache**

Si le module `mod_rewrite` d'Apache est installé sur le serveur web local, le nom du contrôleur frontal `index.php/` peut être retiré de l'URL. Ceci est rendu possible grâce aux règles de réécriture d'URL configurées dans le fichier `web/.htaccess`.

REMARQUE **Configurer Jobeet sur un serveur Microsoft Windows IIS**

L'installation et la configuration d'un projet Symfony est légèrement différente sur les serveurs IIS des environnements Windows. Le tutoriel figurant à l'adresse http://www.symfony-project.org/cookbook/1_0/en/web_server_iis explique pas à pas comment configurer Symfony sur ce type de serveur.

Figure 1–4
Page d'accueil par défaut d'un projet Symfony en environnement de production

Figure 1–5
Page d'accueil par défaut d'un projet Symfony en environnement de développement

Contrôler le code source avec Subversion

Quels sont les avantages d'un gestionnaire de versions ?

C'est une bonne pratique d'utiliser un logiciel de contrôle de versions des fichiers source lorsque l'on développe une application web. En effet, l'utilisation de tels logiciels de suivi de versions assure aux développeurs de nombreux avantages comme :

- travailler avec confiance puisqu'il n'y a plus aucun risque de perdre le moindre fichier source du projet étant donné que Subversion sauvegarde tout ;
- revenir à une version antérieure si un changement casse une portion de code quelconque ;
- travailler efficacement à plusieurs sur le même projet en évitant les conflits ;
- avoir accès à toutes les versions successives de l'application.

Cette dernière section décrit comment utiliser Subversion avec Symfony. Les utilisateurs d'un autre outil de suivi de versions tels que CVS ou GIT pourront s'inspirer de la démarche présentée ici pour l'adapter à leur logiciel de contrôle de code source. Cette nouvelle étape considère qu'un serveur Subversion est déjà installé sur la machine et configuré pour être accessible depuis le protocole HTTP. Par conséquent, seul le processus de création du dépôt Subversion est décrit.

ASTUCE Utiliser un service gratuit en ligne de suivi du code source

Lorsqu'il est impossible d'avoir accès à un dépôt Subversion, des solutions alternatives existent. En effet, des services gratuits en ligne comme Google Code ou GIT Hub permettent aux développeurs de se créer gratuitement leur propre dépôt de code.

Installer et configurer le dépôt Subversion

La première étape d'installation d'un dépôt Subversion consiste d'abord à créer un répertoire dédié au projet Jobeet dans le dépôt global du serveur Subversion.

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

Puis, sur la machine, la structure de base du dépôt Subversion doit être créée. Celle-ci inclut entre autres les répertoires pour gérer le tronc, les branches et les tags du projet. Tout le cycle de vie du projet Jobeet se passera dans le tronc du dépôt.

```
$ svn mkdir -m "created default directory structure"
  ➥ http://svn.example.com/jobeet/trunk
  ➥ http://svn.example.com/jobeet/tags
  ➥ http://svn.example.com/jobeet/branches
```

Il convient ensuite d'extraire le contenu vide du répertoire `trunk/` à l'aide de la commande `svn checkout`.

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

L'étape suivante consiste à vider le contenu des répertoires `cache/` et `log/` du projet étant donné qu'ils n'ont aucun intérêt à être présents dans le dépôt Subversion.

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

À présent, il faut s'assurer que les permissions en écriture sont bien définies sur les répertoires `cache/` et `log/` afin que le serveur web puisse écrire dans chacun d'eux.

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

L'ajout des fichiers dans le dépôt peut maintenant être effectué à l'aide de la commande `svn add`. Les fichiers ne sont pas encore envoyés au serveur Subversion. Ils sont juste marqués comme prêts à être envoyés et sauvegardés dans le dépôt.

```
$ svn add *
```

Étant donné qu'aucun fichier ne devra être enregistré dans les répertoires `cache/` et `log/` du dépôt, il convient d'ajouter le contenu de ces répertoires à la liste de fichiers ignorés par Subversion.

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

L'éditeur de texte par défaut configuré pour SVN devrait se lancer. Subversion doit absolument ignorer tout le contenu de ce répertoire. Par conséquent, il suffit de saisir le caractère étoile dans l'éditeur de texte.

```
*
```

Pour valider la modification, le fichier doit être sauvegardé et l'éditeur de texte fermé. Cette opération doit à son tour être répétée pour le répertoire `log/`.

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

De la même manière, le caractère étoile doit être saisi.

```
*
```

Il ne reste finalement plus qu'à valider tous ces changements et à les envoyer au serveur Subversion afin qu'il se charge de les sauvegarder et de les versionner. Pour ce faire, un simple appel à la commande `svn import` suffit comme le montre le code ci-dessous.

```
$ svn import -m "made the initial import"
  ➥. http://svn.example.com/jobeet/trunk
```

> **ASTUCE Gérer un dépôt Subversion à l'aide d'un client graphique**
>
> Les utilisateurs de Windows peuvent s'appuyer sur l'excellent logiciel TortoiseSVN pour gérer leurs dépôts Subversion en toute simplicité dans un explorateur de fichiers graphique.

En résumé…

Ce tout premier chapitre s'achève ici. Bien qu'il n'ait pas encore été véritablement question de Symfony, le projet Jobeet a pu néanmoins démarrer dans de bonnes conditions et repose déjà sur des bases solides. En effet, la première application Symfony générée est déjà sécurisée par défaut, le serveur web est configuré proprement, un dépôt Subversion a été installé afin de suivre les évolutions du code source du projet, et enfin d'autres bonnes pratiques de développement web ont été présentées. Par conséquent, le projet est prêt à recevoir ses premières lignes de code.

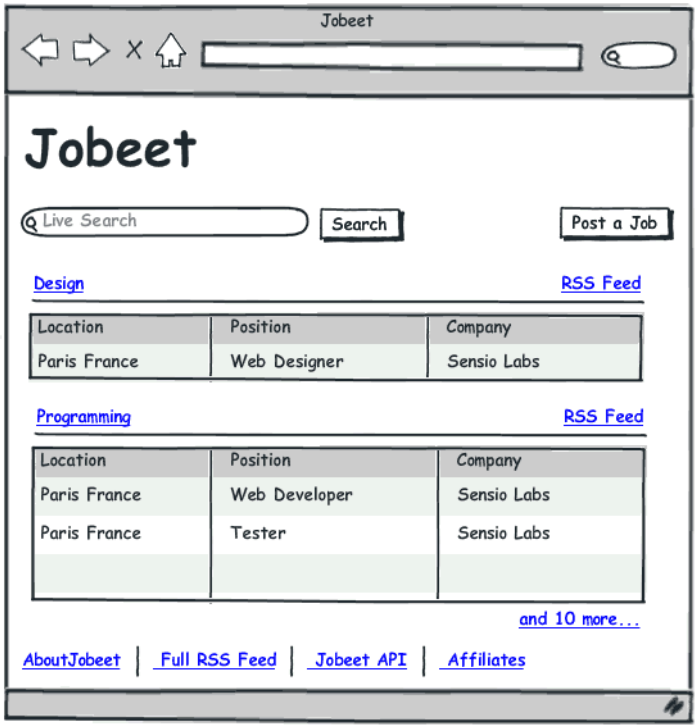
Le chapitre qui suit ne s'intéresse pas encore au code puisqu'il sera uniquement question d'y révéler les différentes spécifications fonctionnelles du projet qui seront implémentées à partir du troisième chapitre.

L'ensemble du projet Jobeet a été versionné dans un dépôt Subversion (http://svn.jobeet.org/doctrine/) du site officiel de Symfony. Ce dépôt contient le code source de l'application à chaque chapitre, ce qui permet de le récupérer à n'importe quelle étape de son avancée. Par exemple, pour récupérer tous les fichiers source du premier chapitre, il suffit d'extraire la version marquée `release_day_01` à l'aide de la commande `svn checkout`.

```
$ svn co http://svn.jobeet.org/doctrine/tags/release_day_01/ jobeet/
```


symfony

chapitre 2


Jobeet
Jobeet
Live Search
Search
Post a Job
Design
RSS Feed
Location
Position
Company
Paris France
Web Designer
Sensio Labs
Programming
RSS Feed
Location
Position
Company
Paris France
Web Developer
Sensio Labs
Paris France
Tester
Sensio Labs
and 10 more...
AboutJobeet
Full RSS Feed
Jobeet API
Affiliates

L'étude de cas

Tout projet professionnel doit démarrer avec une étude préalable des besoins fonctionnels, menant ensuite à l'élaboration de documents de spécifications techniques.

Ce second chapitre est a pour seul objectif de présenter les besoins fonctionnels de l'étude de cas qui sera développée tout au long de cet ouvrage, en ayant recours à des descriptions simples et des maquettes graphiques d'interface.

MOTS-CLÉS :

- Framework Symfony
- maquette d'interface graphique
- Étude des besoins fonctionnels

Tout projet professionnel informatique qui se respecte est élaboré suivant un planning jalonné en plusieurs étapes. Parmi ces étapes figurent obligatoirement l'analyse des besoins fonctionnels du commanditaire ainsi que la rédaction de spécifications techniques. Ces deux étapes sont bien évidemment menées par les équipes décisionnelles en amont du développement du projet afin de valider ce que les équipes de production devront réaliser pour satisfaire les besoins du client et des utilisateurs finaux de l'application.

L'étude de cas qui sera développée pas à pas au cours de cet ouvrage bénéficie elle aussi de spécifications fonctionnelles, dans le but de déterminer l'ensemble des fonctionnalités majeures de l'application. Par conséquent, ce chapitre se destine exclusivement à présenter la liste des besoins fonctionnels à l'aide de maquettes d'interface graphique. Deux sections distinctes seront spécifiées : l'application grand public, le *frontend* et l'interface d'administration du site, le *backend*.

À la découverte du projet…

Le premier chapitre de cet ouvrage n'était résolument pas orienté vers le développement des premières lignes de code PHP, puisqu'il s'agissait de préparer le terrain en installant l'environnement de développement, puis de créer un projet vide avec Symfony. Les premiers pas avec la ligne de commande de Symfony ont également permis de s'assurer que le projet, qui sera dévoilé dans les sections suivantes, repose déjà sur des bases solides et qu'il est convenablement configuré avec des paramètres de sécurité par défaut. En progressant un peu au-delà des explications du premier chapitre, le lecteur aura sûrement remarqué l'écran de félicitations de Symfony qui confirme que le projet est prêt à démarrer.

À l'heure où cet ouvrage est rédigé, une crise économique touche toute la planète depuis plusieurs mois. Le licenciement des salariés ne cesse quant à lui de croître dans de nombreux secteurs d'activité… Heureusement les développeurs Symfony ont la chance de ne pas véritablement se sentir concernés par la crise, et c'est probablement pour cette raison que l'apprentissage de Symfony se justifie. Néanmoins aujourd'hui, il est encore particulièrement difficile de trouver des développeurs Symfony très compétents.

Les questions qui se posent alors sont les suivantes : où peut-on trouver des développeurs Symfony ? Et comment les développeurs peuvent-ils promouvoir leurs compétences avec Symfony ?

Figure 2–1
Page d'accueil par défaut d'un nouveau projet Symfony

Il faut pour cela trouver un gestionnaire d'offres d'emploi qui se focalise uniquement sur les annonces. Ce gestionnaire doit rendre possible la recherche des meilleures personnes qualifiées dans leur domaine d'expertise respectif. Ce cyberespace doit être un lieu convivial où il est facile et rapide de rechercher une offre d'emploi ou d'en proposer une nouvelle.

Inutile de chercher plus loin ! *Jobeet* est l'application idéale pour ce genre de besoin. Jobeet est un logiciel Open Source de gestion d'offres d'emploi qui ne fait qu'une seule chose, mais qui la fait bien. Il est facile à utiliser, à personnaliser, à faire évoluer et bien sûr à embarquer dans d'autres applications web. Par ailleurs, il supporte nativement plusieurs langues, et fait bien évidemment usage des toutes dernières technologies Web 2.0 innovantes afin d'améliorer l'expérience utilisateur. Il fournit également des flux RSS ainsi qu'une API pour interagir avec lui par le biais de services web.

Ce genre d'applications n'existe-t-il pas déjà sur Internet aujourd'hui ? En tant qu'utilisateur, il ne sera pas difficile de trouver des gestionnaires d'offres d'emploi comme Jobeet sur Internet. Le plus compliqué est toutefois d'essayer d'en trouver un qui soit à la fois libre et disposant de nombreuses fonctionnalités riches comme celles qui seront développées ici. Enfin, le dernier avantage de Jobeet est qu'il suffit de moins de 24 heures pour le développer avec Symfony. Dans cet ouvrage, il est question de 21 chapitres à lire et à pratiquer à son rythme…

ASTUCE **Trouver de nouveaux développeurs grâce à Symfonians**

Il existe depuis déjà quelques années un véritable outil Open Source de gestion d'offres d'emploi sur Internet à destination des recruteurs et développeurs Symfony : le site symfonians.net (http://symfonians.net).

Découvrir les spécifications fonctionnelles de Jobeet

Les différents acteurs et applications impliqués

Avant de plonger dans le code, il est important de décrire davantage les spécificités du projet. Les sections qui suivent décrivent les fonctionnalités qui seront implémentées à la première version (itération) de Jobeet. Ces fonctionnalités ont été établies à partir de quelques cas d'utilisation.

Le site Internet de Jobeet possède quatre types d'acteurs :

- *l'administrateur* qui a les pleins pouvoirs sur le site Internet, c'est-à-dire qui peut tout y faire ;
- *l'utilisateur* qui se contente de visiter le site Internet à la recherche d'un emploi ;
- *le posteur* (ou recruteur) qui visite le site afin de poster une nouvelle offre d'emploi ;
- *l'affilié* qui relaie quelques offres d'emploi sur son propre site Internet.

Le projet se décompose également en deux applications distinctes. La première est l'interface Internet grand public, appelée *frontend*, à partir de laquelle les utilisateurs interagissent. Elle leur permet entre autres de consulter et de déposer de nouvelles offres d'emploi, de s'inscrire et d'utiliser l'API de services web, ou encore de s'abonner à des flux RSS. L'application frontend est décrite dans les cas d'utilisation F1 à F7.

La seconde application est l'interface d'administration, autrement dénommée *backend*, dans laquelle les administrateurs ont la possibilité de gérer l'intégralité des contenus dynamiques comme les utilisateurs, les offres d'emploi, les catégories ou encore les affiliés. Cette zone est bien évidemment sécurisée et strictement réservée aux utilisateurs possédant les droits d'accès requis pour y pénétrer. Les fonctionnalités de cette application sont décrites dans les cas d'utilisations B1 à B4.

Utilisation de l'application grand public : le frontend

Scénario F1 : voir les dernières offres en page d'accueil

Lorsqu'un utilisateur arrive sur le site Internet de Jobeet, il découvre une liste des dernières offres d'emploi actives. Les annonces sont d'abord classées par catégorie par ordre alphabétique croissant, puis par date de publication décroissante. Pour chaque annonce, seuls le type de poste, la société et sa localisation sont affichés.

Pour chaque catégorie, la liste montre seulement les dix premières offres et un lien permet de lister toutes les annonces pour la catégorie donnée (*se reporter au cas d'utilisation F2*).

Depuis la page d'accueil, ou tout autre page, l'utilisateur peut affiner la liste des offres d'emploi (*cas d'utilisation F3*) ou poster une nouvelle annonce sur le site (*cas d'utilisation F5*).

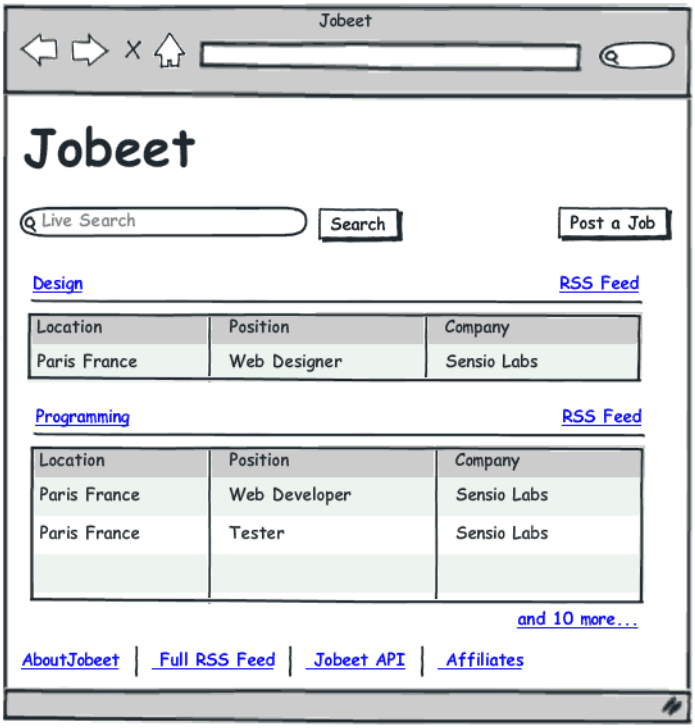


Figure 2–2
Maquette de la page d'accueil de Jobeet pour le scénario F1

Scénario F2 : voir les offres d'une catégorie

Lorsqu'un utilisateur clique sur le nom de la catégorie ou bien sur le lien *more jobs* depuis la page d'accueil, il accède à la liste de toutes les offres de cette catégorie, classées par date de publication décroissante. Afin de faciliter la navigation et l'expérience utilisateur, la liste est paginée avec vingt annonces maximum par page.

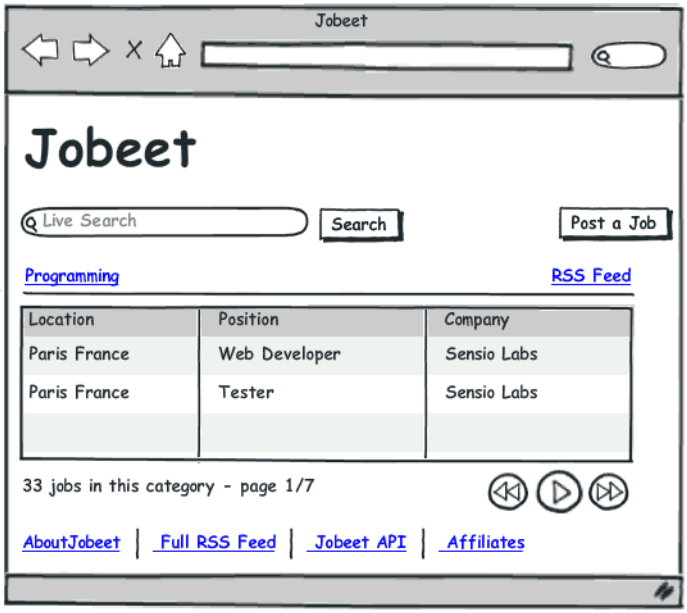


Figure 2–3
Maquette de la liste des offres d'une catégorie pour le scénario F2

Scénario F3 : affiner la liste des offres avec des mots-clés

Comme toute application web dynamique hébergeant du contenu en masse, il est important de faciliter la remontée d'informations à partir d'un moteur de recherche.

L'utilisateur peut ainsi saisir une série de mots-clés pour affiner sa recherche et réduire le nombre d'offres à celles qui correspondent le mieux à ses attentes. Les mots-clés saisis par ce dernier peuvent être des informations issues des champs *localisation*, *type de poste*, *nom de la catégorie* ou encore *nom de la société*.

L'utilisation de la technologie client JavaScript assure également un meilleur confort d'utilisation et une expérience utilisateur accrue en réduisant en temps réel la sélection d'offres d'emploi à chaque fois que l'utilisateur tape un nouveau caractère dans le moteur de recherche. Pour des raisons d'accessibilité, cette fonctionnalité est développée avec du code non intrusif pour garantir un fonctionnement en mode « dégradé » lorsque le JavaScript n'est pas activé sur le poste de l'utilisateur.

Scénario F4 : obtenir le détail d'une offre

L'utilisateur peut sélectionner une offre d'emploi en cliquant sur le type de poste depuis la liste, afin d'obtenir l'intégralité des informations la concernant. La page de détails affiche les informations suivantes de l'annonce :

- le type de poste à pourvoir ;
- le nom de la société ;
- le logo de la société ;
- le lien vers le site de la société ;
- la localisation du poste ;
- le type de contrat (temps plein, temps partiel ou freelance) ;
- la description du poste ;
- et la démarche à suivre pour postuler.

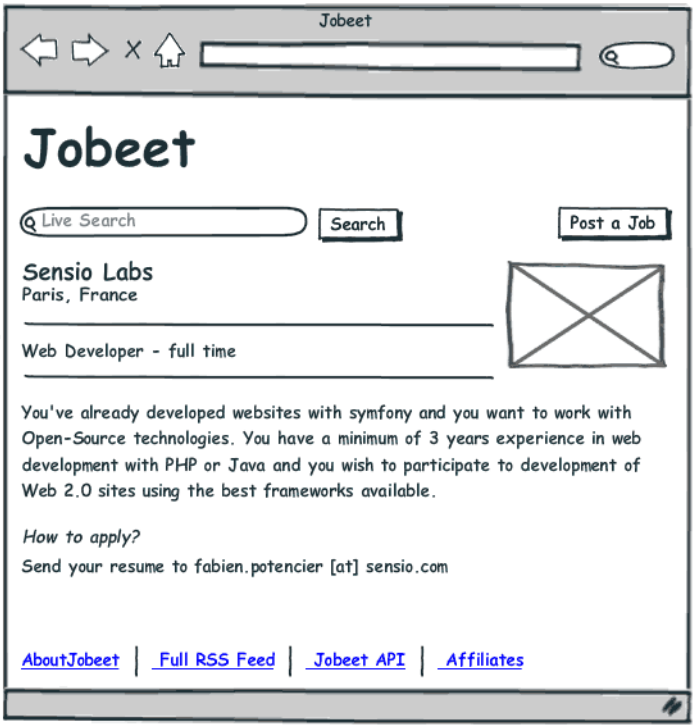


Figure 2–4
Maquette du détail d'une offre d'emploi pour le scénario F4

Scénario F5 : poster une nouvelle annonce

Un utilisateur peut librement ajouter une nouvelle offre d'emploi au site Internet. Cette dernière se compose de plusieurs types d'information, dont certains sont obligatoires :

- le nom de la société ;
- le type de contrat (temps plein, temps partiel ou freelance) ;
- le logo de la société (optionnel) ;
- l'URL du site de la société (optionnel) ;
- le type de poste ;

- la localisation ;
- le nom de la catégorie (choisi d'après une liste de catégories possibles) ;
- la description de l'offre (les adresses e-mails et les URLs sont automatiquement transformées en liens cliquables) ;
- la démarche à suivre pour postuler (les adresses e-mails et les URLs sont automatiquement transformées en liens cliquables) ;
- le mode de diffusion qui indique si l'offre peut être publiée ou pas sur les sites affiliés ;
- l'adresse e-mail de l'auteur de l'annonce.

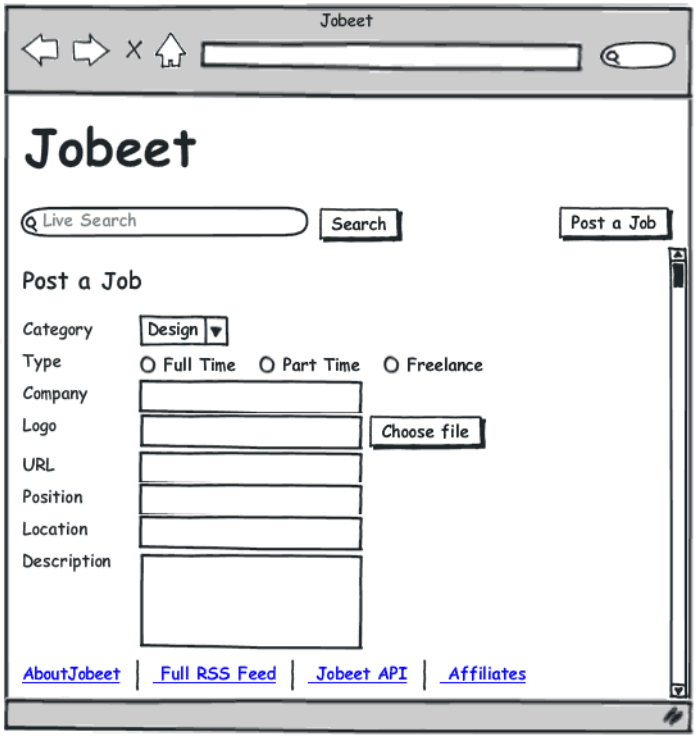


Figure 2–5
Maquette du formulaire de création d'une nouvelle offre pour le scénario F5

- Poster une nouvelle offre d'emploi sur Jobeet ne requiert pas de création de compte utilisateur. Le processus d'ajout d'une nouvelle annonce est simple et se réalise en deux temps. Tout d'abord, l'utilisateur remplit le formulaire avec toutes les informations obligatoires pour décrire l'offre, puis le valide en prévisualisant la page de l'annonce finale.

Bien qu'un utilisateur n'ait pas de compte d'abonné sur l'application, il reste en mesure de modifier son annonce plus tard grâce à une URL spécifique dans laquelle figure un jeton unique qui lui est attribué à la création définitive de l'offre.

Chaque annonce dispose d'une durée de vie de trente jours, configurable par l'administrateur du site (*se référer au cas d'utilisation B2*). L'utilisateur peut quant à lui revenir pour réactiver ou prolonger la validité de son offre pour une nouvelle période de trente jours à condition que celle-ci arrive à expiration dans moins de cinq jours.

Scénario F6 : s'inscrire en tant qu'affilié pour utiliser l'API

Un utilisateur a besoin de postuler pour devenir un affilié et être autorisé à manipuler l'API de Jobeet. Pour postuler à l'API, il doit d'abord renseigner les informations suivantes :

- son identité ;
- son adresse e-mail ;
- l'adresse de son site Internet.

Le compte de l'affilié est soumis à la validation expresse des administrateurs (*se référer au cas d'utilisation B4*). Une fois activé, l'affilié reçoit par e-mail le jeton unique qui lui permet d'utiliser l'API. Enfin, quand l'affilié s'abonne au service, il peut aussi choisir un sous-ensemble d'annonces à afficher sur son site Internet en sélectionnant une liste de catégories disponibles dans lesquelles figurent les dernières offres à remonter.

Scénario F7 : l'affilié récupère la liste des dernières offres actives

Un affilié peut récupérer la liste des dernières offres d'emploi actives en appelant l'API à partir de son jeton d'affilié. La liste peut être retournée au choix au format XML, JSON ou bien encore YAML. Cette liste contient alors l'ensemble des informations publiques disponibles d'une annonce.

Utilisation de l'interface d'administration : le backend

Scénario B1 : gérer les catégories

Un administrateur est capable de gérer l'ensemble des catégories du site Internet. Il dispose d'une page lui permettant de lister toutes les catégories disponibles ainsi que d'un écran lui permettant de créer ou d'éditer une catégorie existante. L'application Jobeet étant prévue pour supporter le français et l'anglais, les formulaires de création et d'édition des catégories embarquent les champs de traduction pour chaque langue du site.

Scénario B2 : gérer les offres d'emploi

Un administrateur dispose d'une interface d'administration complète des offres d'emploi publiées sur Jobeet. Un écran de gestion lui permet de lister l'ensemble des annonces classées par date de publication décroissante.

Cette liste se présente sous la forme d'un tableau dont chaque ligne représente une offre. Certains en-têtes du tableau sont cliquables afin de modifier le classement et l'ordre des offres. D'autre part, la liste est paginée avec dix résultats par page et un formulaire de filtres permet de l'affiner en effectuant des recherches sur certains critères prédéfinis.

Chaque offre d'emploi dispose de ses propres actions qui permettent à l'administrateur de l'éditer, de la supprimer ou encore de la prolonger dans le temps. Ces opérations, à l'exception de la fonction d'édition, sont également disponibles sur un lot d'offres sélectionnées au moyen de cases à cocher.

Enfin, une action globale permet à l'utilisateur de nettoyer la base de données des offres d'emploi périmées depuis un certain nombre de jours ou qui n'ont jamais été activées.

Scénario B3 : gérer les comptes administrateur

De la même manière que pour les catégories et les offres d'emploi, l'administrateur dispose d'un panel de gestion des autres administrateurs du site. Ce module lui permet ainsi de lister les personnes autorisées à gérer l'application, mais également de créer de nouveaux comptes ou de supprimer des comptes existants.

Scénario B4 : configurer le site Internet

Enfin, l'administrateur dispose d'un dernier panneau d'administration qui lui permet de gérer l'ensemble des comptes affiliés en attente de validation. Cet espace lui permet en effet d'activer ou de désactiver à sa guise le compte d'un affilié souhaitant profiter de l'API.

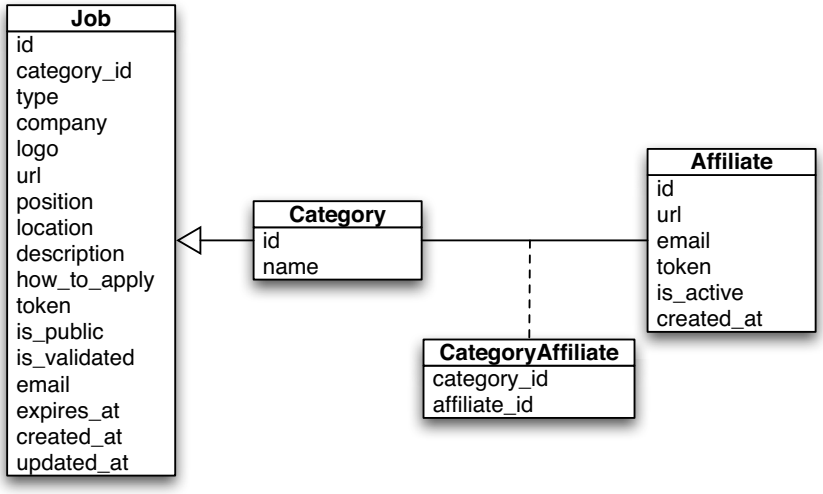
Lorsque le compte d'un nouvel affilié est activé, le système lui crée et lui attribue automatiquement un jeton unique servant d'identifiant sur l'API. Ce jeton lui est envoyé directement par e-mail pour lui indiquer que son compte est valide et en état de marche immédiat.

En résumé…

Dans tout développement web, il est important de ne pas se précipiter sur le codage dès le premier jour. La première étape doit toujours consister en un recueil des besoins du commanditaire, puis en l'écriture des spécifications fonctionnelles et techniques donnant lieu à l'élaboration de maquettes visuelles. C'est exactement ce qu'a voulu montrer ce deuxième chapitre.

Le prochain chapitre entame plus sérieusement le projet en s'intéressant à la création de tout le modèle de la base de données de Jobeet. Il y est notamment question de la prise en main de l'ORM Doctrine, de la génération automatique de la base de données et des classes de modèle, de l'écriture de données initiales de test, de la génération d'un premier module fonctionnel, et bien sûr des premières lignes de code PHP tant attendues…

chapitre 3


Job
id
category_id
type
company
logo
url
position
location
description
how_to_apply
token
is_public
is_validated
email
expires_at
created_at
updated_at
Category
id
name
Affiliate
id
url
email
token
is_active
created_at
CategoryAffiliate
category_id
affiliate_id

Concevoir le modèle de données

La base de données est l'un des piliers de toute application web mais sa nature et sa structure peuvent rendre difficile son intégration dans le développement de l'application. Heureusement, Symfony en simplifie la manipulation grâce à la couche d'ORM embarquée Doctrine, qui automatise la création de la base de données à partir d'un schéma de description et de quelques fichiers de données initiales.

MOTS-CLÉS :

- L'ORM Doctrine
- Base de données
- Programmation orientée objet

REMARQUE Parcimonie du code écrit

Symfony réalise une majeure partie du travail à la place du développeur ; le module web ainsi créé sera entièrement fonctionnel sans que vous ayez à écrire beaucoup de code PHP.

Ce troisième chapitre se consacre principalement à la base de données de Jobeet. Les notions de modèle de données, de couche d'abstraction de bases de données, de librairie d'ORM ou bien encore de génération de code seront abordées. Enfin, le tout premier module fonctionnel de l'application sera développé malgré le peu de code que nous aurons à écrire.

Installer la base de données

Le framework Symfony supporte toutes les bases de données compatibles avec PDO (MySQL, PostgreSQL, SQLite, Oracle, MSSQL…). PDO est la couche native d'abstraction de bases de données de PHP. Dans le cadre de ce projet, c'est MySQL qui a été choisi.

Créer la base de données MySQL

La première étape consiste bien évidemment à créer une base de données locale dans laquelle seront sauvegardées et récupérées les données de Jobeet. Pour ce faire, la commande `mysqladmin` suffit amplement, mais un outil graphique comme `PHPMyAdmin` ou bien `MySQL Query Browser` fait aussi très bien l'affaire.

```
$ mysqladmin –uroot –pmYsEcret create jobeet
```

Le parti pris d'utiliser MySQL pour Jobeet tient juste dans le fait que c'est le plus connu et le plus accessible pour tous. Un autre moteur de base de données aurait pu être choisi à la place dans la mesure où le code SQL sera automatiquement généré par l'ORM. C'est ce dernier qui se préoccupe d'écrire les bonnes requêtes SQL pour le moteur de base de données installé.

Configurer la base de données pour le projet Symfony

Maintenant que la base de données est créée, il faut spécifier à Symfony sa configuration afin qu'elle puisse être reliée à l'application via l'ORM. La commande `configure:database` configure Symfony pour fonctionner avec la base de données :

```
$ php symfony configure:database --name=doctrine
  ➥ --class=sfDoctrineDatabase
  ➥ "mysql:host=localhost;dbname=jobeet" root mYsEcret
```

Après avoir configuré la connexion à la base de données, toutes les connexions qui référencent Propel dans le fichier `config/databases.yml`

doivent être supprimées manuellement. Le fichier de configuration de la base de données doit finalement ressembler à celui-ci :

```
all:
  doctrine:
    class: sfDoctrineDatabase
    param:
      dsn: 'mysql:host=localhost;dbname=jobeet'
      username: root
      password: mYsEcret
```

La tâche `configure:database` accepte trois arguments : le DSN (Data Set Name, lien vers la base de données) PDO, le nom d'utilisateur et le mot de passe permettant d'accéder à la base de données. Si aucun mot de passe n'est requis pour accéder à la base de données du serveur de développement, le troisième argument peut être omis.

REMARQUE **Fichier databases.yml**

La tâche `configure:database` stocke la configuration de la base de données dans le fichier de configuration `config/databases.yml`. Au lieu d'utiliser cette tâche, le fichier peut être édité manuellement.

Présentation de la couche d'ORM Doctrine

Symfony fournit de base deux bibliothèques d'ORM Open-Source pour interagir avec les bases de données : Propel et Doctrine, agissant toutes deux comme des couches d'abstraction. Cependant, cet ouvrage ne s'intéresse qu'à l'utilisation de Symfony avec l'ORM Doctrine.

CHOIX DE CONCEPTION **Pourquoi Doctrine plutôt que Propel ?**

Le choix de la librairie Doctrine par rapport à Propel s'impose de lui-même pour plusieurs raisons. Bien que la librairie Propel soit aujourd'hui mature, il n'en résulte pas moins que son âge lui fait défaut. En effet, ce projet Open-Source rendit bien des services aux développeurs jusqu'à aujourd'hui, mais malheureusement son support et sa communauté ne sont plus aussi actifs qu'auparavant. Propel est clairement sur le point de mourir, laissant place à des outils plus récents comme Doctrine.

La librairie Doctrine dispose de plusieurs atouts par rapport à Propel tels qu'une API simple et fluide pour définir des requêtes SQL, de meilleures performances avec les requêtes complexes, la gestion native des migrations, la validation des données, l'héritage de tables ou bien encore le support de différents comportements utiles (« sluggification », ensembles imbriqués, suppressions virtuelles, recherches…).

De surcroît, le projet Doctrine jouit aujourd'hui d'une communauté toujours plus active et d'une documentation abondante. D'ailleurs, à l'heure où nous écrivons ces lignes, un livre contenant toute la documentation technique de Doctrine est en préparation.

Enfin, le projet Doctrine est supporté par Sensio Labs, société éditrice du framework Symfony. Le développeur principal du projet Doctrine, Jonathan Wage, a rejoint l'équipe de production de la société en 2008 pour se consacrer davantage au développement et à l'intégration de Doctrine dans Symfony.

Qu'est-ce qu'une couche d'abstraction de base de données ?

Une couche d'abstraction de bases de données est une interface logicielle qui permet de rendre indépendant le système de gestion de base de données de l'application. Ainsi, une application fonctionnant sur un système de base de données relationnel (SGBDR), comme MySQL, doit pouvoir fonctionner de la même manière avec un système de base de données différent (Oracle par exemple) sans avoir à modifier son code fonctionnel. Une simple ligne de configuration dans un fichier doit suffire à indiquer que le gestionnaire de base de données n'est plus le même.

Depuis la version 5.1.0, PHP dispose de sa propre couche d'accès aux bases de données : PDO. PDO est l'abréviation pour « PHP Data Objects ». Il s'agit en fait surtout d'une interface commune d'accès aux bases de données plus qu'une véritable couche d'abstraction. En effet, avec PDO, il est nécessaire d'écrire soi-même les requêtes SQL pour interroger la base de données à laquelle l'application est connectée. Or, les requêtes sont largement dépendantes du système de base de données, bien que SQL soit un langage standard et normalisé. Chaque SGBDR propose en réalité ses propres fonctionnalités, et donc sa propre version enrichie de SQL pour interroger la base de données.

Doctrine s'appuie sur l'extension PDO pour tout ce qui concerne la connexion et l'interrogation des bases de données (requêtes préparées, transactions, ensembles de résultats…). En revanche, l'API se charge de convertir les requêtes SQL pour le système de gestion de base de données actif, ce qui en fait une véritable couche d'abstraction.

La librairie Doctrine supporte toutes les bases de données compatibles avec PDO telles que MySQL, PostgreSQL, SQLite, Oracle, MSSQL, Sybase, IBM DB2, IBM Informix…

Qu'est-ce qu'un ORM ?

ORM est le sigle de « Object-Relational Mapping » ou « Mapping Objet Relationnel » en français. Une couche d'ORM est une interface logicielle qui permet de représenter et de manipuler sous forme d'objet tous les éléments qui composent une base de données relationnelle.

Ainsi, une table ou bien un enregistrement de celle-ci est perçu comme un objet du langage sur lequel il est possible d'appliquer des actions (les méthodes). L'avantage de cette approche est de s'abstraire complètement de la technologie de gestion de la base de données qui fonctionne en arrière-plan, et de ne travailler qu'avec des objets ayant des liaisons entre eux.

À partir d'un modèle de données défini plus loin dans ce chapitre, Doctrine construit entièrement la base de données pour le SGBDR choisi,

ainsi que les classes permettant d'interroger la base de données au travers d'objets. Grâce aux relations qui lient les tables entre elles, Doctrine est par exemple capable de retrouver tous les enregistrements d'une table qui dépendent d'un autre dans une seconde table.

Activer l'ORM Doctrine pour Symfony

Les deux ORMs du framework, Propel et Doctrine, sont tous deux fournis nativement sous forme de plug-ins internes. À ce jour, Propel est encore la couche d'ORM activée par défaut dans la configuration d'un projet Symfony. Il faut donc commencer par le désactiver, puis activer le plug-in `sfDoctrinePlugin` qui contient toute la bibliothèque Doctrine. La manipulation est triviale puisqu'elle ne nécessite qu'une unique modification dans le fichier de configuration générale du projet `config/ProjectConfiguration.class.php` comme le montre le code suivant :

```
public function setup()
{
  $this->enablePlugins(array('sfDoctrinePlugin'));
  $this->disablePlugins(array('sfPropelPlugin'));
}
```

Le même résultat peut également être obtenu en une seule ligne de code. Le listing ci-dessous active par défaut tous les plug-ins du projet, hormis ceux spécifiés dans le tableau passé en paramètre.

```
public function setup()
{
  $this->enableAllPluginsExcept(array('sfPropelPlugin',
'sfCompat10Plugin'));
}
```

L'une ou l'autre de ces deux opérations nécessite de vider le cache du projet Symfony.

```
$ php symfony cache:clear
```

D'autre part, certains plug-ins comme `sfDoctrinePlugin` embarquent des ressources supplémentaires qui doivent être accessibles depuis un navigateur web comme des images, des feuilles de style ou bien encore des fichiers JavaScript. Lorsqu'un plug-in est nouvellement installé et activé, l'exécution de la tâche `plugin:publish-assets` crée les liens symboliques qui permettent de publier toutes les ressources web nécessaires.

```
$ php symfony plugin:publish-assets
```

Comme Propel n'est pas utilisé pour ce projet Jobeet, le lien symbolique vers le répertoire `web/sfPropelPlugin` qui subsiste peut être supprimé en toute sécurité.

```
$ rm web/sfPropelPlugin
```

Il est temps à présent de s'intéresser à l'architecture de la base de données qui accueille toutes les données de Jobeet.

Concevoir le modèle de données

Le chapitre précédent a décrit les cas d'utilisation de l'application Jobeet ainsi que tous les composants nécessaires : les offres d'emploi, les affiliations et les catégories. Néanmoins, cette définition des besoins fonctionnels n'est pas suffisante. Une transposition sous forme d'un diagramme UML apporte plus de visibilité sur chaque objet et les relations qui les lient les uns aux autres.

Découvrir le diagramme UML « entité-relation »

D'après l'étude des besoins fonctionnels de Jobeet, on détermine clairement les différentes relations suivantes :

- une offre d'emploi est associée à une catégorie ;
- une catégorie a entre 0 et N offres d'emploi associées ;
- une affiliation possède entre 1 et N catégories ;
- une catégorie a entre 0 et N affiliations.

Il en résulte presque naturellement le modèle « entité-relation » suivant.

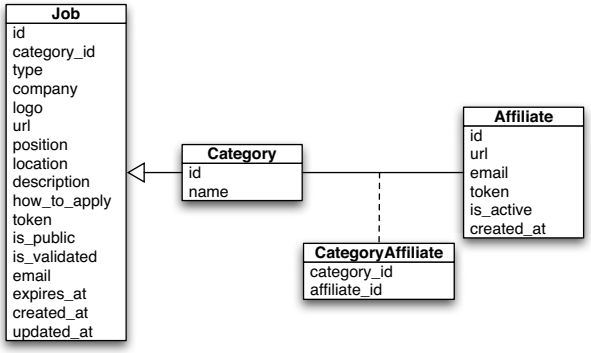


Figure 3–1
Diagramme entité-relation de Jobeet

Ce diagramme décrit également les différentes propriétés de chaque objet qui deviendront les noms des colonnes de chaque table de la base

de données. Un champ `created_at` a été ajouté à quelques tables. Symfony reconnaît ce type de champ et fixe sa valeur avec la date courante du serveur lorsqu'un enregistrement est créé. Il en va de même pour les champs `updated_at` : leur valeur est définie à partir de la date courante du serveur lorsque l'enregistrement est mis à jour dans la table.

Mise en place du schéma de définition de la base

De l'importance du schéma de définition de la base de données…

Les offres d'emploi, les affiliations et les catégories doivent être stockées dans la base de données relationnelle installée plus haut. Symfony est un framework qui a la particularité d'être entièrement orienté Objet, ce qui permet au développeur de manipuler des objets aussi souvent que possible. Par exemple, au lieu d'écrire des requêtes SQL pour retrouver des enregistrements de la base de données, il sera plus naturel et logique de manipuler des objets.

Dans Symfony, les informations de la base de données relationnelle sont représentées (« mappées » en langage informatique) en un modèle objet. La génération et la gestion de modèles objets sont entièrement laissées à la charge de l'ORM Doctrine. Pour ce faire, Doctrine a besoin d'une description des tables et de leurs relations pour créer toutes les classes correspondantes. Il existe deux manières pour établir ce schéma de description. La première consiste à analyser une base de données existante par rétro-ingénierie (*reverse engineering* pour les puristes) ou bien en le créant manuellement.

Écrire le schéma de définition de la base de données

Comme la base de données n'existe pas encore et que nous souhaitons la garder agnostique, le schéma de définition de la base de données doit être écrit à la main dans le fichier `config/doctrine/schema.yml`. Le répertoire `config/doctrine/` n'existe pas encore, il doit être créé à la main.

```
$ mkdir config/doctrine
$ touch config/doctrine/schema.yml
```

Le fichier `config/doctrine/schema.yml` contient une définition au format YAML de tous les éléments qui composent la base de données. Cette description indique la structure de chaque table et de ses champs respectifs, les contraintes appliquées sur chacun d'eux ainsi que toutes les relations qui les lient les unes aux autres. Le code suivant correspond au schéma de définition de la base de données de l'application Jobeet.

Contenu du fichier 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: { 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 }
```

Le schéma est la traduction directe, en format YAML, du diagramme « entité-relation ». On identifie ici clairement quatre entités dans ce modèle : `JobeetCategory`, `JobeetJob`, `JobeetAffiliate` et `JobeetCategoryAffiliate`. Les relations entre les objets `JobeetCategory` et `JobeetJob` découvertes plus haut sont bien retranscrites au même titre que celles entre `JobeetCategory` et `JobeetAffiliate` via la quatrième entité `JobeetCategoryAffiliate`.

De plus, ce modèle définit des comportements pour certaines entités grâce à la section `actAs`. Les comportements (*behaviors* en anglais) sont des outils internes de Doctrine qui permettent d'automatiser des traitements sur les données lorsqu'elles sont écrites dans la base. Ici, le comportement `Timestampable` permet de créer et de fixer les valeurs des champs `created_at` et `updated_at` à la volée par Doctrine.

Si les tables de la base de données sont directement créées grâce aux requêtes SQL ou bien à l'aide d'un éditeur graphique, le fichier de configuration `schema.yml` correspondant peut être construit en exécutant la tâche `doctrine:build-schema`.

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

FORMAT YAML pour la sérialisation des données

D'après le site officiel de YAML, YAML est « un standard de sérialisation des données, facile à utiliser pour un être humain quel que soit le langage de programmation ».
En d'autres termes, YAML est un langage simple pour décrire des données (chaîne de caractères, entiers, dates, tableaux ou tableaux associatifs).
En YAML, la structure est présentée grâce à l'indentation. Les listes d'éléments sont identifiées par un tiret, et les paires clé/valeur d'une section par une virgule. YAML dispose égalementd'une syntaxe raccourcie pour décrire la même structure en moins de lignes. Les tableaux sont explicitement identifiés par des crochets [] et les tableaux associatifs avec des accolades {}.
Si vous n'êtes pas familier avec YAML, c'est le moment de commencer à vous y intéresser dans la mesure où le framework Symfony l'emploie excessivement pour ses fichiers de configuration.
Il y a enfin une chose importante dont il faut absolument se souvenir lorsque l'on édite un fichier YAML : l'indentation doit toujours être composée d'un ou de plusieurs espaces, mais jamais de tabulations.

Déclaration des attributs des colonnes d'une table en format YAML

Le fichier `schema.yml` contient la description de toutes les tables et de leurs colonnes. Chaque colonne est décrite au moyen des attributs suivants :

1. `type` : le type de la colonne (`float`, `decimal`, `string`, `array`, `object`, `blob`, `clob`, `timestamp`, `time`, `date`, `enum`, `gzip`) ;
2. `notnull` : placé à la valeur `true`, cet attribut rend la colonne obligatoire ;

À PROPOS Comportements supportés par Doctrine

L'attribut `onDelete` détermine le comportement `ON DELETE` des clés étrangères. Doctrine supporte les comportements `CASCADE`, `SET NULL` et `RESTRICT`. Par exemple, lorsqu'un enregistrement de la table `job` est supprimé, tous les enregistrements associés à la table `jobeet_category_affiliate` seront automatiquement effacés de la base de données.

3 `unique` : placé à la valeur `true`, l'attribut `unique` crée automatiquement un index d'unicité sur la colonne.

La base de données existe et est configurée pour fonctionner avec Symfony, et le schéma de description de cette dernière est désormais écrit. Néanmoins, la base de données est toujours vierge et rien ne permet pour le moment de la manipuler. La partie suivante couvre ces problématiques en présentant comment l'ORM Doctrine génère tout le nécessaire pour rendre la base de données opérationnelle.

Générer la base de données et les classes du modèle avec Doctrine

Grâce à la description de la base de données présente dans le fichier `config/doctrine/schema.yml`, Doctrine est capable de générer les ordres SQL nécessaires pour créer les tables de la base de données ainsi que toutes les classes PHP qui permettent de l'attaquer au travers d'objets métiers.

Construire la base de données automatiquement

La construction de la base de données est réalisée en trois temps : la génération des classes du modèle de données, puis la génération du fichier contenant toutes les requêtes SQL à exécuter, et enfin l'exécution de ce dernier pour créer physiquement toutes les tables.

La première étape consiste tout d'abord à générer le modèle de données, autrement dit les classes PHP relatives à chaque table et enregistrement de la base de données.

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

Cette commande génère un ensemble de fichiers PHP dans le répertoire `lib/model/doctrine` qui correspondent à une entité du schéma de définition de la base de données.

Lorsque toutes les classes du modèle de données sont prêtes, l'étape suivante doit permettre de générer tous les scripts SQL qui créent physiquement les tables dans la base de données. Cette fois encore, Symfony facilite grandement le travail du développeur grâce à la tâche automatique `doctrine:build-sql` qu'il suffit d'exécuter.

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

La tâche `doctrine:build-sql` crée les requêtes SQL dans le répertoire `data/sql/`, optimisées pour le moteur de base de données configuré :

Échantillon de code du fichier 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;
```

Enfin, il ne reste plus que la dernière étape à franchir. Il s'agit de créer physiquement toutes les tables dans la base de données. Une fois de plus, c'est un jeu d'enfant grâce aux tâches automatiques fournies par le framework. Le plug-in `sfDoctrinePlugin` comporte une tâche `doctrine:insert-sql` qui se charge d'exécuter le script SQL généré précédemment pour monter toute la base de données.

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

Ça y est, la base de données est prête à accueillir des informations. Toutes les tables ont été créées ainsi que les contraintes d'intégrité référentielle qui lient les enregistrements des tables entre eux. Il est désormais temps de s'intéresser aux classes du modèle qui ont été générées.

À PROPOS Aide sur les tâches automatiques

Comme n'importe quel outil en ligne de commande, les tâches automatiques de Symfony peuvent prendre des arguments et des options. Chaque tâche est livrée avec un manuel d'utilisation qui peut être affiché grâce à la commande `help`.

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

Le message d'aide liste tous les arguments et options possibles, donne la valeur par défaut de chacun d'eux, et donne quelques exemples pratiques d'utilisation.

Découvrir les classes du modèle de données

À la première étape de construction de la base de données, les fichiers PHP du modèle de données ont été générés à l'aide de la tâche `doctrine:build-model`. Ces fichiers correspondent aux classes PHP qui transforment les enregistrements d'une table en objets métiers pour l'application.

La tâche `doctrine:build-model` construit les fichiers PHP dans le répertoire `lib/model/doctrine/` qui permettent d'interagir avec la base de données. En parcourant ces derniers, il est important de remarquer que Doctrine génère trois classes par table. Par exemple, pour la table `jobeet_job` :

1. `JobeetJob` : un objet de cette classe représente un seul enregistrement de la table `jobeet_job`. La classe est vide par défaut ;
2. `BaseJobeetJob` : c'est la superclasse de `JobeetJob`. Chaque fois que l'on exécute la tâche `doctrine:build-model`, cette classe est régénérée, c'est pourquoi toutes les personnalisations doivent être écrites dans la classe `JobeetJob` ;
3. `JobeetJobTable` : la classe définit des méthodes qui retournent principalement des collections d'objets `JobeetJob`. Cette classe est vide par défaut.

Les valeurs des colonnes d'un enregistrement sont manipulées à partir d'un objet modèle en utilisant quelques accesseurs (méthodes `get*()`) et mutateurs (méthodes `set*()`) :

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

echo $job->getPosition();

$job->delete();
```

Au vu de cette syntaxe, il est évident que manipuler des objets plutôt que des requêtes SQL devient à la fois plus naturel, plus aisé mais aussi plus sécurisé, puisque l'échappement des données est laissé à la charge de l'ORM.

Le modèle de données de Jobeet établit une relation entre les offres d'emploi et les catégories. Au moment de générer les classes du modèle de données, Doctrine a deviné les relations possibles entre les entités et les a reportées fidèlement dans les classes PHP générées. Ainsi, un objet `JobeetJob` dispose de méthodes pour définir ou bien récupérer l'objet `JobeetCategory` qui lui est associé comme le présente l'exemple de code ci-dessous.

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

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

Doctrine fonctionne bilatéralement, c'est-à-dire que les liaisons entre les objets sont gérées aussi bien d'un côté que d'un autre. La classe `JobeetJob` possède des méthodes pour agir sur l'objet `JobeetCategory` qui lui est associé mais la classe `JobeetCategory` possède elle aussi des méthodes pour définir les objets `JobeetJob` qui lui appartiennent.

Générer la base de données et le modèle en une seule passe

La tâche `doctrine:build-all` est un raccourci pour les tâches exécutées dans cette section et bien d'autres. Il est temps maintenant de générer les formulaires et les validateurs pour les classes de modèle de Jobeet.

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

Les validateurs seront présentés à la fin de ce chapitre, tandis que les formulaires seront expliqués en détail au cours du chapitre 10.

Symfony charge automatiquement les classes PHP à la place du développeur, ce qui signifie que nul appel à `require` n'est requis dans le code.

C'est l'une des innombrables fonctionnalités que le framework automatise, bien que cela entraîne un léger inconvénient. En effet, chaque fois qu'une nouvelle classe est ajoutée au projet, le cache de Symfony doit être vidé. La tâche `doctrine:build-model` a généré un certain nombre de nouvelles classes, c'est pourquoi le cache doit être réinitialisé.

```
$ php symfony cache:clear
```

Le développement d'un projet est fortement accéléré grâce aux nombreux composants et tâches automatiques que fournit nativement le framework Symfony. Dans le cas présent, la base de données ainsi que toutes les classes PHP du modèle de données ont été mises en place en un temps record. Imaginez le temps qu'il vous aurait fallu pour réaliser tout cela à la main en partant de rien !

Toutefois, il manque encore quelque chose d'essentiel pour pouvoir se lancer pleinement dans le code et le développement des fonctionnalités de Jobeet. Il s'agit bien sûr des données initiales qui permettent à l'application de s'initialiser et d'être testée. La section suivante se consacre pleinement à ce sujet.

ASTUCE Comprendre la syntaxe des tâches automatiques de Symfony

Une tâche Symfony est constituée d'un espace de nom (*namespace* pour les puristes) et d'un nom. Chacun d'eux peut être raccourci tant qu'il n'y a pas d'ambiguïté avec d'autres tâches. Ainsi, les commandes suivantes sont équivalentes à `cache:clear` :

```
$ php symfony cache:cl
$ php symfony ca:c
```

Comme la tâche `cache:clear` est fréquemment utilisée, elle possède une autre abréviation encore plus courte :

```
$ php symfony cc
```

Préparer les données initiales de Jobeet

Les tables ont été créées dans la base de données mais celles-ci sont toujours vides. Il faut donc préparer quelques jeux de données pour remplir les tables de la base de données au fur et à mesure de l'avancée du projet.

Découvrir les différents types de données d'un projet Symfony

Pour n'importe quelle application, il existe trois types de données :

1. *les données initiales* : ce sont les données dont a besoin l'application pour fonctionner. Par exemple, Jobeet requiert quelques catégories. S'il n'y en a pas, personne ne pourra soumettre d'offre d'emploi. Un utilisateur administrateur capable de s'authentifier à l'interface d'administration (*backend* en anglais) est également nécessaire ;
2. *les données de test* : les données de test sont nécessaires pour tester l'application et pour s'assurer qu'elle se comporte comme les cas d'utilisation fonctionnels le spécifient. Bien évidemment, le meilleur moyen de le vérifier est d'écrire des séries de tests automatisés ; c'est pourquoi des tests unitaires et fonctionnels seront développés pour Jobeet. Ainsi, à chaque fois que les tests seront exécutés, une base de données saine constituée de données fraîches et prêtes à être testées sera nécessaire ;

3 *les données utilisateur* : les données utilisateur sont créées par les utilisateurs au cours du cycle de vie normal de l'application.

Pour le moment, Jobeet requiert quelques données initiales pour initialiser l'application. Symfony fournit un moyen simple et efficace de définir ce type de données à l'aide de fichiers YAML, comme l'explique la partie suivante.

Définir des jeux de données initiales pour Jobeet

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 bien exécuter quelques requêtes SQL avec le programme MySQL. Néanmoins, comme ce besoin est relativement fréquent, il existe une meilleure façon de procéder avec Symfony. Il s'agit de créer des fichiers YAML dans le répertoire `data/fixtures/`, puis d'exécuter la tâche `doctrine:data-load` pour les charger en base de données.

Les deux listings suivants de code YAML définissent un jeu de données initiales pour l'application. Le premier fichier déclare les données pour remplir la table `jobeet_category` tandis que le second sert à peupler la table des offres d'emploi `jobeet_job`.

Contenu du fichier data/fixtures/categories.yml

```
JobeetCategory:
  design:
    name: Design
  programming:
    name: Programming
  manager:
    name: Manager
  administrator:
    name: Administrator
```

Contenu du fichier 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'
```

Un fichier de données est écrit au format YAML, et décrit les objets modèles référencés par un nom unique. Par exemple, les deux offres d'emploi sont intitulées `job_sensio_labs` et `job_extreme_sensio`. Cet intitulé sert à lier les objets entre eux sans avoir à exprimer explicitement les clés primaires, qui sont dans la plupart des cas auto-incrémentées et qui varient perpétuellement. La catégorie de l'offre d'emploi `job_sensio_labs` est `programming`, ce qui correspond à la catégorie nommée « Programming ».

Dans un fichier YAML, lorsqu'une chaîne de caractères contient des retours à la ligne (comme la colonne `description` dans les données initiales du fichier d'offres d'emploi), la barre verticale (*pipe* en anglais) | sert à indiquer que la chaîne occupera plusieurs lignes.

REMARQUE Télécharger les images relatives aux données initiales

Le fichier de données initiales des offres référence 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

Vous devrez ensuite les placer dans le répertoire `uploads/jobs/`.

ASTUCE **Propel versus Doctrine**

Propel a besoin que les fichiers de données de test soient préfixés par des nombres pour déterminer dans quel ordre les fichiers doivent être chargés. Avec Doctrine, ce n'est pas nécessaire puisque toutes les données sont chargées et sauvegardées dans le bon ordre pour s'assurer que les clés étrangères sont définies correctement.

Bien qu'un fichier de données contienne des objets provenant d'un ou de plusieurs modèles, il est vivement recommandé de ne créer qu'un seul fichier par modèle.

Dans un fichier de données, il n'est nul besoin de définir toutes les valeurs des colonnes. Si certaines valeurs ne sont pas définies, Symfony utilisera la valeur par défaut définie dans le schéma de la base de données. Comme Symfony utilise Doctrine pour charger les données en base de données, tous les comportements natifs (comme la fixation automatique des colonnes `created_at` et `updated_at`) et les comportements personnalisés ajoutés aux classes de modèle sont activés.

Charger les jeux de données de tests en base de données

Une fois les fichiers de données initiales créés, leur chargement en base de données est aussi simple que de lancer une tâche automatique. Le plug-in `sfDoctrinePlugin` possède la commande `doctrine:data-load` qui se charge d'enregistrer toutes ces données dans la base de données.

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

Exécuter l'une après l'autre toutes les tâches pour régénérer la base de données, construire les classes du modèle et insérer les données initiales peut se révéler très vite fastidieux. Symfony propose une tâche simple qui réalise toutes ces opérations en une seule passe comme l'explique la section suivante.

Régénérer la base de données et le modèle en une seule passe

La tâche `doctrine:build-all-reload` est un raccourci pour la `tâche doctrine:build-all` suivi de la tâche `doctrine:data-load`. Celle-ci s'occupe de régénérer toute la base de données et les classes du modèle, puis finit par charger les données initiales dans les tables.

```
$ php symfony doctrine:build-all-reload
```

Il suffit de lancer la commande `doctrine:build-all-reload` puis de s'assurer que tout a bien été généré depuis le schéma. Cette tâche génère les classes de formulaires, de filtres, de modèle, supprime la base de données existante et la recrée avec toutes les tables peuplées par les données initiales.

Profiter de toute la puissance de Symfony dans le navigateur

L'interface en ligne de commande est plutôt pratique mais n'est malgré tout pas très attrayante, qui plus est pour un projet web. À ce stade d'avancement du projet, Jobeet est déjà prêt à accueillir les pages web dynamiques qui interagissent avec la base de données.

La suite du chapitre aborde les fonctionnalités essentielles d'affichage de la liste des offres d'emploi, et d'édition et de suppression d'une offre existante. Comme cela a déjà été expliqué au premier chapitre, un projet Symfony est constitué d'*applications*. Chaque application est ensuite divisée en *modules*.

Un module est un ensemble autonome de code PHP qui représente une fonctionnalité de l'application (le module API par exemple), ou bien un ensemble de manipulations que l'utilisateur peut réaliser sur un objet du modèle (un module d'offres d'emploi par exemple).

Générer le premier module fonctionnel « job »

Le module principal de l'application Jobeet est bien évidemment celui qui permet de créer et de consulter des offres. Le framework Symfony est capable de générer automatiquement un module fonctionnel complet pour un modèle donné. Ce module intègre de base toutes les fonctionnalités de manipulation simples telles que l'ajout, la modification, la suppression et la consultation.

Ce travail est réalisé à l'aide de la commande `doctrine:generate-module` comme le présente le code ci-dessous.

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

La tâche `doctrine:generate-module` génère un module `job` dans l'application `frontend` pour le modèle `JobeetJob`. Comme avec la plupart des tâches Symfony, quelques fichiers et répertoires ont été créés. Tous les fichiers du présent module ont été fabriqués sous le répertoire `apps/frontend/modules/job/`.

Composition de base d'un module généré par Symfony

Le tableau ci-après décrit les répertoires de base qui ont été générés par Symfony à l'exécution de la tâche `doctrine:generate-module` dans le répertoire `apps/frontend/modules/job/`.

Tableau 3–1 Répertoires du module apps/frontend/modules/job

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

Le répertoire `actions/` contient la classe dans laquelle se trouvent toutes les actions CRUD (*create*, *retrieve*, *update* et *delete*) de base qui permettent de manipuler une offre d'emploi. À toutes ces actions est associé un ensemble de fichiers de templates générés dans le répertoire `templates/`.

Découvrir les actions du module « job »

Le fichier `actions/actions.class.php` définit toutes les actions possibles pour le module `job`. C'est exactement ce que décrit le tableau 3-2.

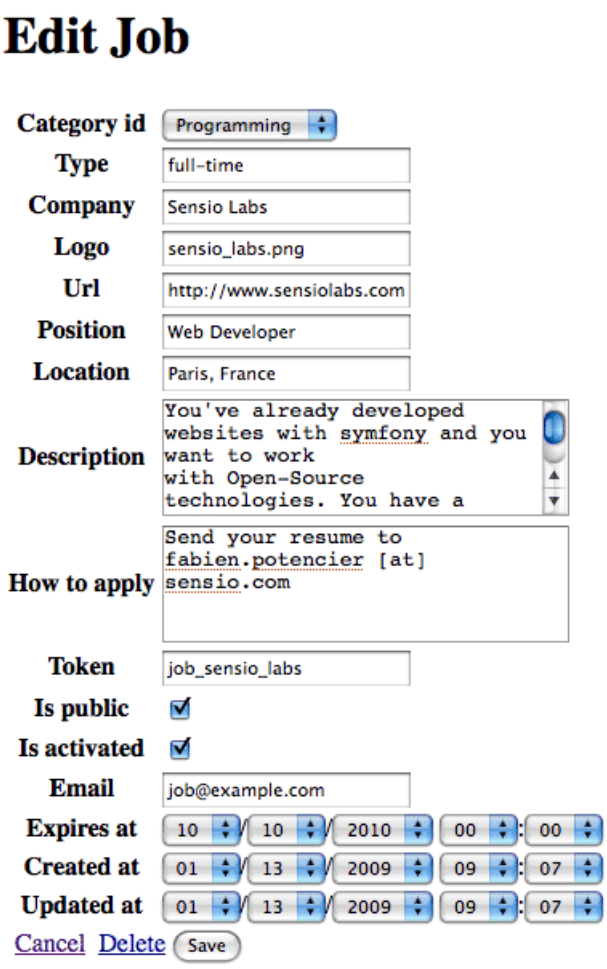


Figure 3–2
Formulaire d'édition d'une offre d'emploi

Tableau 3–2 Liste des actions du fichier apps/frontend/modules/job/actions/actions.class.php

Nom de l'action	Description
`index`	Affiche les enregistrements d'une table
`show`	Affiche les champs et les valeurs d'un enregistrement donné
`new`	Affiche un formulaire pour créer un nouvel enregistrement
`create`	Crée un nouvel enregistrement
`edit`	Affiche un formulaire pour éditer un enregistrement existant
`update`	Met à jour les informations d'un enregistrement d'après les valeurs transmises par l'utilisateur
`delete`	Supprime un enregistrement donné de la table

Le module `job` est désormais accessible et utilisable depuis un navigateur web à l'adresse suivante :

- http://jobeet.localhost/frontend_dev.php/job

Comprendre l'importance de la méthode magique __toString()

Si l'on tente d'éditer une offre d'emploi, on remarque que la liste déroulante « Category id » est une sélection de l'ensemble des catégories présentes dans la base de données. La valeur de chaque option est obtenue grâce à la méthode `__toString()`.

Doctrine essaye d'appeler nativement une méthode `__toString()` en devinant un nom de colonne descriptif tel que `title`, `name`, `subject`, etc. Si l'on désire quelque chose de plus personnalisé, il est alors nécessaire de redéfinir la méthode `__toString()` comme le présente le listing ci-après. Le modèle `JobeetCategory` est capable de deviner la méthode `__toString()` en utilisant la colonne `name` de la table `jobeet_category`.

Listing du fichier 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());
  }
}
```

Listing du fichier lib/model/doctrine/JobeetAffiliate.class.php

```
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function __toString()
  {
    return $this->getUrl();
  }
}
```

Ajouter et éditer les offres d'emploi

Les offres d'emploi sont à présent prêtes à être ajoutées et éditées. Si un champ obligatoire est laissé vide ou bien si sa valeur est incorrecte (une date invalide par exemple), le processus de validation du formulaire provoquera une erreur, empêchant alors la mise à jour de l'enregistrement. Symfony crée effectivement les règles de validation basiques en introspectant le schéma de la base de données.

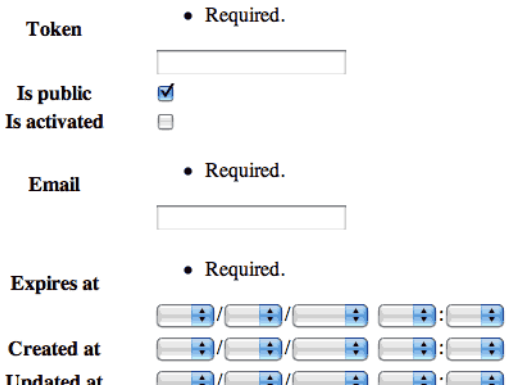


Figure 3–3
Contrôle de saisie basique dans le formulaire de création d'une offre d'emploi.

En résumé…

C'est tout pour ce troisième chapitre. L'introduction était très claire. En effet, peu de code PHP a été écrit mais Jobeet dispose déjà d'un module d'offres d'emploi entièrement fonctionnel et prêt à être amélioré et personnalisé. Souvenez-vous, moins de code PHP signifie aussi moins de risques d'y trouver un bug !

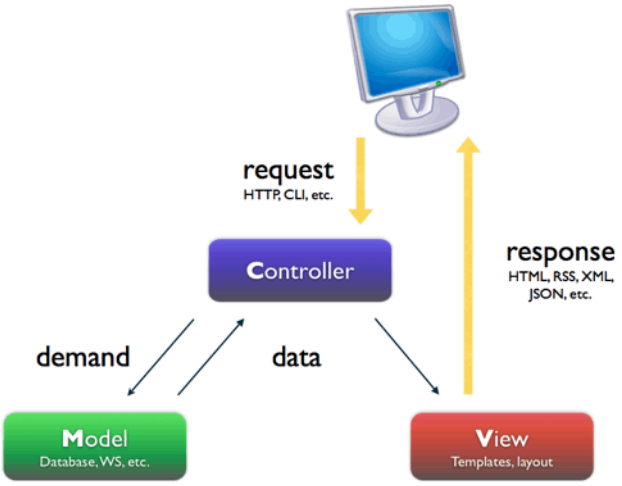
Un moyen simple de progresser avant de passer au chapitre suivant est de prendre la peine de lire le code généré pour le module et le modèle, et essayer d'en comprendre le fonctionnement.

Le chapitre qui suit aborde l'un des plus importants paradigmes utilisés dans les frameworks web : le patron de conception MVC.

Le code complet du chapitre est disponible dans le dépôt SVN de Jobeet au tag `release_day_03` :

```
$ svn co http://svn.jobeet.org/doctrine/tags/release_day_03/
jobeet/
```

chapitre 4


request
HTTP, CLI, etc.
Controller
response
HTML, RSS, XML,
JSON, etc.
demand
data
Model
Database, WS, etc.
View
Templates, layout

Le contrôleur et la vue

Structurer une application web n'est pas toujours évident du fait des nombreux composants qui interviennent au cours du développement. Le paradigme modèle, vue, contrôleur est l'une des solutions capables de répondre à ce besoin. Nous verrons ainsi comment Symfony intègre parfaitement ce motif de conception éprouvé dans un projet.

MOTS-CLÉS :

- Paradigme Modèle Vue Contrôleur
- Objets sfWebRequest et sfWebResponse
- Layout

Le chapitre précédent a montré comment Symfony simplifie la gestion d'une base de données en abstrayant les différences entre les moteurs de base de données et en convertissant des objets relationnels en classes PHP. De plus, ce fut l'occasion de découvrir la couche d'ORM Doctrine qui permet d'automatiser la création d'une base de données à partir d'un schéma de description et de quelques fichiers de données initiales.

Ce chapitre s'intéresse à la personnalisation du module basique d'offres d'emploi généré précédemment. Ce module dispose déjà de tout le code nécessaire pour l'application :

- une page qui liste toutes les offres d'emploi ;
- une page pour créer une nouvelle offre d'emploi ;
- une page pour mettre à jour une offre d'emploi existante ;
- une page pour supprimer une offre d'emploi.

L'architecture MVC et son implémentation dans Symfony

Développer un site en PHP sans recourir à un framework signifie en général de n'avoir qu'un seul fichier PHP par page HTML, chaque fichier ayant la même structure : initialisation et configuration générale, logique métier relative à la page appelée, récupération des enregistrements de la base de données, et enfin le code HTML qui construit la page.

Même en utilisant un moteur de templates pour séparer la logique métier du code HTML final, ou en recourant à une couche d'abstraction pour séparer les interactions du modèle de la base de données, il n'en résulte pas moins, la plupart du temps, une trop grosse quantité de code qui se révèle être un véritable cauchemar à maintenir.

Heureusement, à chaque problème sa solution. En développement web, la solution actuelle la plus répandue pour organiser du code de manière efficace et maintenable est d'avoir recours au motif de conception MVC. Ce dernier définit en effet un moyen d'organiser le code en fonction de la nature de chacune de ses parties. Ce patron sépare ainsi le code en trois couches distinctes :

- le **modèle** qui définit la logique métier (la base de données appartient au modèle). Dans Symfony, toutes les classes et tous les fichiers propres au modèle sont stockés dans le répertoire `lib/model/` ;
- la **vue** qui est l'interface avec quoi l'utilisateur interagit (un moteur de templates fait partie de la vue). Dans Symfony, la couche vue est principalement constituée de templates PHP. Ils sont stockés dans les différents répertoires `templates/` ;

- le **contrôleur** qui est la partie du code appelant le modèle pour en récupérer des données qu'il transmet ensuite à la vue pour le rendu final au client. Lors de l'installation de Symfony le premier jour, il a été montré que toutes les requêtes étaient gérées par les *front controllers* (`index.php` et `frontend_dev.php`). Ces contrôleurs frontaux délèguent le véritable travail aux **actions**. Ces dernières sont logiquement groupées à l'intérieur de **modules**.

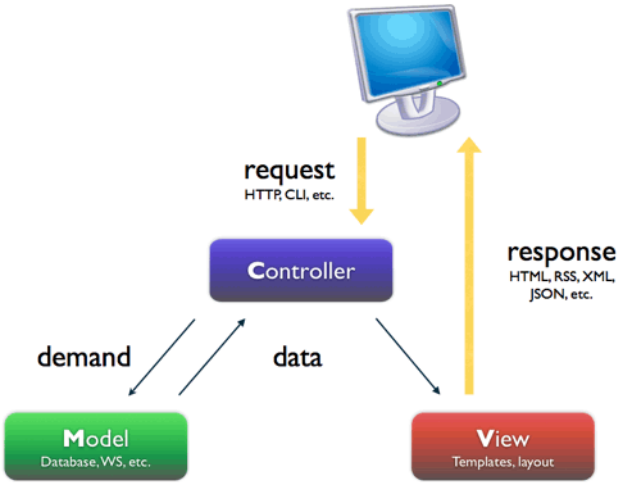


Figure 4–1
Schéma de fonctionnement du motif de conception MVC

Les prochaines pages de cet ouvrage s'appuient sur les maquettes (*mock-ups* en anglais) établies au second chapitre. La page d'accueil et la page des offres seront personnalisées et dynamisées. Dans la foulée, de nombreuses améliorations seront apportées dans différents fichiers, afin de présenter la structure de fichiers de Symfony et la manière de séparer le code entre les différentes couches.

Habiller le contenu de chaque page avec un même gabarit

Décorer une page avec un en-tête et un pied de page

Pour commencer, en regardant de plus près les maquettes, on identifie clairement que la plupart des pages HTML se ressemblent. Or, il a été démontré juste avant qu'il vaut mieux éviter à tout prix la duplication de code, qu'il s'agisse de code HTML ou bien de code PHP. Mais comment empêcher cette copie des éléments communs de la vue ? Un moyen

BONNES PRATIQUES **Le principe « DRY »**

DRY est un acronyme pour *Don't Repeat Yourself*. Il s'agit d'une philosophie en développement informatique qui consiste à limiter le code redondant (i.e. la duplication). De cette manière, le débogage et la maintenance s'en voient grandement simplifiés.

simple de résoudre ce problème est de définir un en-tête et un pied de page, puis de les inclure dans chaque template.

Figure 4–2
Structure d'une page web en trois parties

Décorer le contenu d'une page avec un décorateur

Malheureusement, ici, l'en-tête et le pied de page ne contiennent pas de code HTML valide. Il faut donc opter pour une meilleure manière de faire. Au lieu de réinventer la roue, Symfony s'appuie sur un autre motif de conception : le patron Décorateur (*decorator* en anglais). Ce dernier résout le problème autrement, en habillant le contenu rendu par le template principal, appelé **layout**.

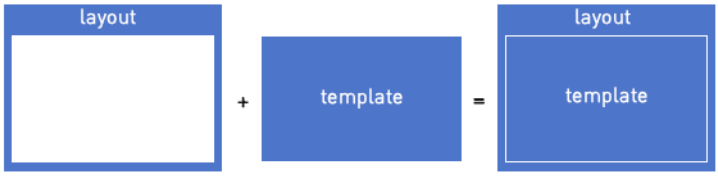


Figure 4–3
Schéma de fonctionnement du motif de conception Décorateur

Dans Symfony, le layout par défaut d'une application est un fichier PHP appelé `layout.php` et se trouve dans le répertoire `apps/frontend/templates/`. Celui-ci contient l'ensemble des templates globaux d'une application.

Listing du fichier 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="/job">
            <img src="/images/jobeet.gif" alt="Jobeet Job Board" />
          </a></h1>
```

```
    <div id="sub_header">
      <div class="post">
        <h2>Ask for people</h2>
        <div>
          <a href="/job/new">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>
```

Ce layout fait appel à des fonctions et fait référence à des variables PHP. La variable `$sf_content` est l'une des plus importantes car elle est automatiquement définie par le framework lui-même et contient le code HTML généré par l'action.

Désormais, en parcourant le module job (http://jobeet.localhost/frontend_dev.php/job), toutes les actions sont décorées par le layout.

Intégrer la charte graphique de Jobeet

Récupérer les images et les feuilles de style

Ce livre ne s'intéresse pas au design web, c'est pourquoi toutes les ressources nécessaires au projet ont été préparées à l'avance. Les images et les feuilles de style sont toutes disponibles en téléchargement.

1. Téléchargez l'archive des images (http://www.symfony-project.org/get/jobeet/images.zip) et placez les dans le répertoire `web/images/` ;
2. Téléchargez l'archive des feuilles de style (http://www.symfony-project.org/get/jobeet/css.zip), puis placez les dans le répertoire `web/css/`.

Le fichier `layout.php` fait appel à un « favicon ». Le favicon de Jobeet peut être téléchargé à l'adresse http://www.symfony-project.org/get/jobeet/favicon.ico puis déposé à la racine du répertoire `web/`.

Par défaut, la tâche `generate:project` crée trois répertoires pour les ressources du projet : `web/images/` pour les images, `web/css/` pour les feuilles de style, et `web/js/` pour les JavaScripts. C'est l'une des conventions définies par Symfony, mais il est bien sûr possible de les stocker ailleurs dans le répertoire `web/`.

Le lecteur assidu aura remarqué que même si le fichier `main.css` est mentionné nulle part dans le layout par défaut, il est finalement présent dans le code HTML généré. Comment est-ce possible ?

La feuille de style a été incluse par l'appel de la fonction `include_stylesheets()`, se trouvant dans le tag `<head>` du layout. La fonction `include_stylesheets()` est en fait un **helper**.

DÉFINITION Helpers dans Symfony

Un helper est une fonction définie par Symfony qui peut prendre des paramètres et qui renvoie du code HTML. La plupart du temps, les helpers embarquent des petits bouts de code fréquemment utilisés dans les templates et font ainsi gagner du temps.

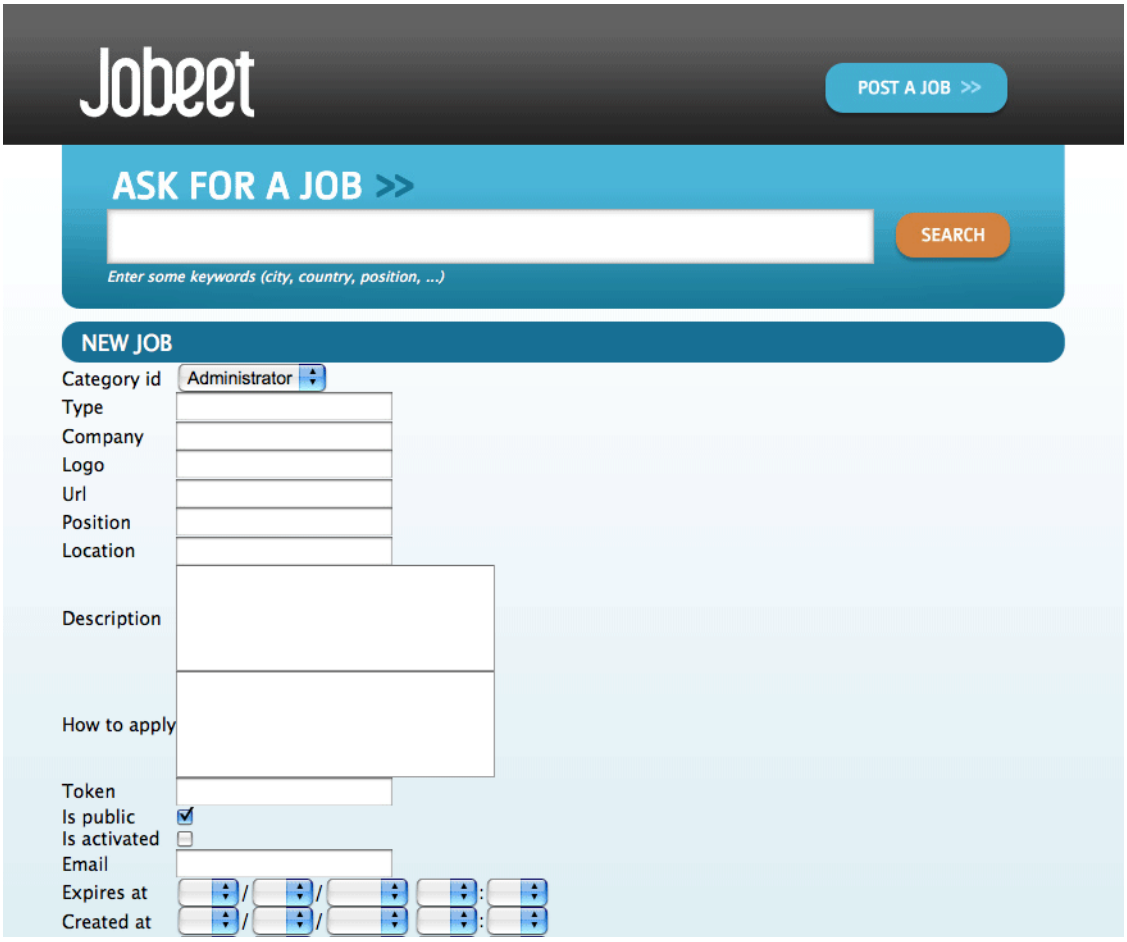


Figure 4–4 Interface graphique de Jobeet

Configurer la vue à partir d'un fichier de configuration

Comment le helper a-t-il connaissance des feuilles de style à inclure dans la page ?

La couche Vue peut être modelée en éditant le fichier de configuration de l'application `view.yml`. Ci-dessous, le contenu de celui généré par défaut par la tâche `generate:app` :

Fichier de configuration de la vue, généré par défaut : 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 de tous les templates de l'application. Par exemple, la section des feuilles de style définit un tableau de fichiers de feuilles de style à inclure pour chaque page de l'application. Dans le layout, l'inclusion se fait au moyen du helper `include_stylesheets()`.

Dans le fichier de configuration `view.yml`, la feuille de style référencée par défaut est `main.css`, et non `/css/main.css`. En fait, les deux définitions sont équivalentes puisque Symfony préfixe les chemins relatifs par `/css/`.

Si plusieurs fichiers sont définis, le framework les inclura tous dans l'ordre de leur déclaration :

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

L'attribut `media` est aussi modifiable et le suffixe `.css` peut être omis.

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

Cette configuration génère le code HTML 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 de l'application. Par défaut, le nom est `layout`, et donc Symfony décore chaque page avec le fichier `layout.php`. Le processus de décoration peut également être désactivé en fixant l'entrée `has_layout` à `false`.

Cela marche tel quel mais le fichier `jobs.css` est toujours nécessaire pour la page d'accueil tandis que le fichier `job.css` est seulement requis pour

la page de détail d'une offre. Le fichier de configuration `view.yml` est personnalisable pour chaque module de base.

Le bout de code ci-dessous configure le fichier `view.yml` de l'application afin qu'il ne fasse appel qu'au fichier `main.css`.

Extrait du fichier apps/frontend/config/view.yml

```
stylesheets:    [main.css]
```

Pour personnaliser la vue du module d'offres d'emploi, il suffit de créer un nouveau fichier `view.yml` dans le répertoire `apps/frontend/modules/job/config/` :

Contenu du fichier apps/frontend/modules/job/config/view.yml

```
indexSuccess:
  stylesheets: [jobs.css]

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

Les sections `indexSuccess` et `showSuccess` correspondent aux noms des templates des actions `index` et `show` qui seront évoquées plus tard. Chaque constante de configuration se trouvant sous la section `all` peut être redéfinie dans les nouvelles sections créées. La section spéciale `all` permet de définir les paramètres de configuration partagés par l'ensemble des actions du module.

Configurer la vue à l'aide des helpers de Symfony

En règle générale, tout ce qu'il est possible de paramétrer dans un fichier de configuration peut être accompli de la même manière avec du code PHP. Par exemple, au lieu de créer un fichier `view.yml` spécifique au module `job`, nous pouvons directement inclure une feuille de style depuis un template à l'aide du helper `use_stylesheet()` :

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

Utiliser ce helper dans le fichier `layout.php` pour charger des feuilles de style revient à inclure cette dernière de manière globale pour chaque module de l'application.

Choisir l'une ou l'autre des deux méthodes est en fait une simple question de goût. Le fichier `view.yml` fournit une manière de définir des paramètres pour toutes les actions d'un module, ce qui est impossible dans un template. Par ailleurs, le fichier de configuration est totalement statique. En revanche, avoir recours au helper `use_stylesheet()` est bien

IMPORTANT Principes de configuration dans Symfony

Pour les différents fichiers de configuration de Symfony, le même paramètre peut être modifié à plusieurs niveaux :

- la configuration par défaut dans le framework ;
- la configuration globale du projet (dans `config/`) ;
- la configuration locale d'une application (dans `apps/APP/config/`) ;
- la configuration locale restreinte à un module (dans `apps/APP/modules/MODULE/config/`).

En cours d'exécution, le système de configuration fusionne toutes les valeurs des différents fichiers s'ils existent, et met en cache le résultat pour de meilleures performances.

plus flexible puisque chaque chose est à sa place, la définition des feuilles de style comme le code HTML.

Tout au long de cet ouvrage, c'est le helper `use_stylesheet()` qui sera utilisé. Le fichier `apps/frontend/modules/job/config/view.yml`, qui vient tout juste d'être ajouté, peut finalement être supprimé au profit de l'emploi du helper `use_stylesheet()` dans les templates du module `job`.

Helper à ajouter en haut du fichier apps/frontend/modules/job/templates/indexSuccess.php

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

Helper à ajouter en haut du fichier apps/frontend/modules/job/templates/showSuccess.php

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

De la même manière, la configuration des JavaScripts est réalisée grâce à la section `javascripts` du fichier de configuration `view.yml` ou bien grâce à l'appel du helper `use_javascript()` directement dans un template.

Générer la page d'accueil des offres d'emploi

Comme il l'a été présenté au chapitre 3, la page d'accueil des offres d'emploi est générée par l'action `index` du module `job`. L'action `index` est la partie Contrôleur de la page, et le template associé, `indexSuccess.php`, est la partie Vue. Le code ci-dessous rappelle la structure arborescente du module `job` généré pour l'application `frontend`.

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

Écrire le contrôleur de la page : l'action index

Chaque action est représentée par une méthode d'une classe. Pour la page d'accueil du module `job`, la classe est `jobActions` (le nom du module suffixé par `Actions`) et la méthode est `executeIndex()` (`execute` suffixé par le nom de l'action). L'action `index` récupère toutes les offres d'emploi de la base de données comme le montre le code ci-dessous.

Contenu du fichier apps/frontend/modules/job/actions/actions.class.php

```
class jobActions extends sfActions
{
  public function executeIndex(sfWebRequest $request)
  {
    $this->jobeet_job_list = Doctrine::getTable('JobeetJob')
      ->createQuery('a')
      ->execute();
  }

  // ...
}
```

Il est temps d'étudier de plus près ces quelques lignes de code. La méthode `executeIndex()` (le contrôleur) appelle la table `JobeetJob` pour créer une requête SQL qui récupère toutes les offres d'emploi. Cette dernière retourne un objet `Doctrine_Collection` – une liste d'objets `JobeetJob` – qui est assigné à la propriété objet `jobeet_job_list`.

Toutes ces propriétés d'objet sont ensuite automatiquement transmises au template (la vue). En résumé, le passage d'une variable du contrôleur à la vue est simple : il suffit de déclarer une nouvelle propriété dans la classe d'actions.

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

Ce code crée les variables `$foo` et `$bar` accessibles dans le template.

Créer la vue associée à l'action : le template

Par défaut, le nom du template associé à une action est déduit par Symfony grâce à une convention : le nom de l'action suffixée par `Success`.

Le template `indexSuccess.php` génère un tableau HTML pour toutes les offres d'emploi. Le code actuel du template est présenté ci-dessous :

Contenu du fichier 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_job_list 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 le code du template, l'instruction `foreach` itère à travers la liste d'objets `JobeetJob` (`$jobeet_job_list`) et, pour chaque offre d'emploi, affiche la valeur des colonnes en sortie. Accéder à la valeur d'une colonne d'une table est aussi simple qu'un appel à une méthode accesseur dont le nom commence par `get`, suivi du nom de la colonne en « Camel Case » (par exemple, la méthode `getCreatedAt()` pour la colonne `created_at`).

Personnaliser les informations affichées pour chaque offre

Le code précédent de la vue affiche toutes les informations de l'objet `JobeetJob`. Néanmoins, toutes ne sont pas forcément pertinentes pour la page d'accueil du site Internet et c'est pour cette raison que seuls la situation géographique, le nom de la société et le type de poste seront affichés.

Contenu du fichier apps/frontend/modules/job/templates/indexSuccess.php

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

<div id="jobs">
  <table class="jobs">
    <?php foreach ($jobeet_job_list 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>
```

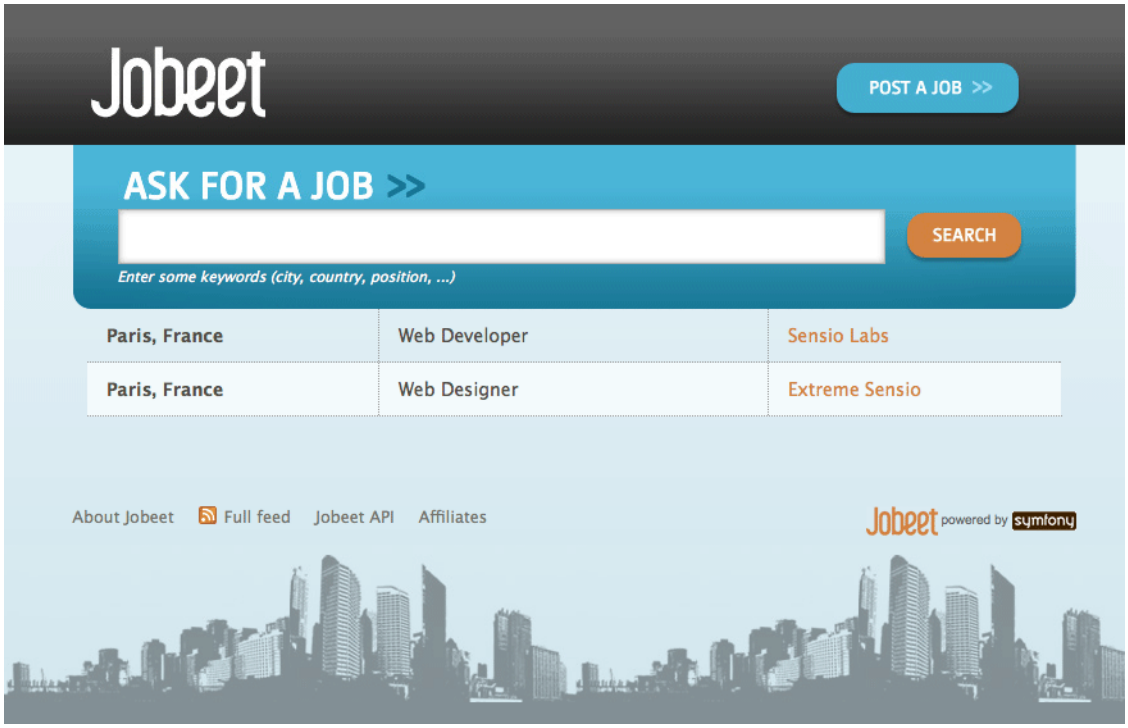


Figure 4–5 Page de listing des offres d'emploi

La fonction `url_for()` appelée dans les templates sera présentée au chapitre suivant. En attendant, il s'agit juste de retenir que ce helper permet de générer des URLs internes ou externes.

Générer la page de détail d'une offre

Créer le template du détail de l'offre

À présent, il est temps de personnaliser le template qui affiche le détail des offres d'emploi. Pour ce faire, nous allons éditer le fichier `showSuccess.php` et remplacer son contenu actuel par le code présenté ci-après.

Contenu du fichier 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 date('m/d/Y',
      ➥ strtotime($job->getCreatedAt())) ?></small>
  </div>

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

Mettre à jour l'action show

Ce template utilise la variable `$job`, transmise par l'action, pour afficher les informations détaillées d'une offre d'emploi. La variable passée au template a été renommée de `$jobeet_job` en `$job` ; ce même changement doit donc être opéré sur les deux occurrences de la variable présentes dans le corps de l'action `show`.

Détail de la méthode executeShow() du fichier 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);
}
```

La description d'une offre d'emploi est formatée à l'aide du helper `simple_format_text()`. Celui-ci remplace les retours à la ligne par des balises `<br/>`. Ce helper appartient au groupe des helpers `Text` qui ne sont pas chargés par défaut par le framework. L'appel au helper `use_helper()` permet de charger manuellement tous les helpers du groupe `Text` et de les rendre disponibles dans le template.

Figure 4–6
Page de détail d'une offre d'emploi

Utiliser les emplacements pour modifier dynamiquement le titre des pages

Pour le moment, le titre de toutes les pages est inscrit en dur dans le tag `<title>` du layout :

Extrait du fichier apps/frontend/templates/layout.php

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

Bien évidemment, il serait plus judicieux de déterminer un titre plus explicite, composé du nom de la société et de l'intitulé du poste, pour chaque page détaillant une offre d'emploi.

Dans Symfony, lorsqu'une zone du layout dépend du template à afficher, il devient nécessaire de déclarer un emplacement (*slot*).

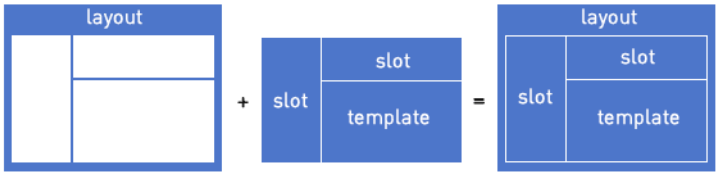


Figure 4–7
Principe de fonctionnement des slots dans Symfony

Ajouter un slot au layout dans la balise `<head>` permet ainsi de définir un titre dynamique pour chaque page.

Définition d'un emplacement pour le titre

Extrait du fichier apps/frontend/templates/layout.php

```
<title><?php include_slot('title') ?></title>
```

Chaque emplacement se définit par un nom (ici `title`) et est affiché au moyen du helper `include_slot()`.

Fixer la valeur d'un slot dans un template

Une fois que le slot est défini, sa valeur peut être fixée depuis n'importe quel template à l'aide du helper `slot()`. Dans le cadre de cette application, il s'agit de définir la valeur du slot « title » afin de modifier dynamiquement le titre de chaque page. Pour ce faire, il suffit de modifier le fichier `showSuccess.php` en lui ajoutant le code ci-dessous.

Définition du titre de la page dans le fichier apps/frontend/modules/job/templates/showSuccess.php

```
<?php slot(
  'title',
  sprintf('%s is looking for a %s', $job->getCompany(), $job-
>getPosition()))
?>
```

Fixer la valeur d'un slot complexe dans un template

Si le titre est complexe à générer, le helper `slot()` peut aussi être employé sous la forme d'un bloc comme le montre le code ci-après.

Exemple de slot complexe dans le fichier 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(); ?>
```

Déclarer une valeur par défaut pour le slot

Pour quelques pages comme la page d'accueil, nous avons seulement besoin d'un titre générique. Au lieu de répéter le même titre encore et encore dans tous les templates, il est possible de déclarer un titre par défaut dans le layout :

Définition d'un titre de page web par défaut apps/frontend/templates/layout.php

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

Le helper `include_slot()` retourne `true` si le `slot` a bien été défini. En somme, lorsque le contenu du slot `title` est fixé depuis un template, il est utilisé ; sinon c'est le titre par défaut qui est retenu.

REMARQUE À propos des helpers

Jusqu'à maintenant, un certain nombre de helpers commençant par `include_` ont été présentés. Ces fonctions génèrent le code HTML et, dans la plupart des cas, ont un helper associé `get_` qui retourne exclusivement le contenu :

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

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

Rediriger vers une page d'erreur 404 si l'offre n'existe pas

La page d'une offre d'emploi est générée au moyen de l'action `show`, déclarée dans la méthode `executeShow()` du module `job` :

Méthode executeShow() de la classe apps/frontend/modules/job/actions/actions.class.php

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

  // ...
}
```

De la même manière que dans l'action `index`, la classe de la table `JobeetJob` permet de retrouver une offre d'emploi, cette fois-ci grâce à la méthode `find()`. Le paramètre de cette méthode est l'identifiant unique d'une offre d'emploi, sa clé primaire. La section suivante expliquera pourquoi le code `$request->getParameter('id')` retourne la clé primaire de l'offre d'emploi.

Si une offre d'emploi n'existe pas dans la base de données, l'idéal est de rediriger l'utilisateur vers une page d'erreur 404. Pour ce faire, il suffit d'utiliser la méthode `forward404Unless()`. Cette dernière prend un booléen comme premier argument et, s'il n'est pas à `true`, arrête le flot d'exécution en cours. Les méthodes `forward` arrêtent immédiatement l'exécution de l'action en cours en lançant une exception `sfError404Exception`, ce qui explique qu'il n'est pas nécessaire de retourner de valeur ensuite.

Comme pour toutes les exceptions lancées, la page affichée à l'utilisateur est différente en fonction de l'environnement (production ou développement) :

Figure 4–8
Page d'erreur 404 en environnement de développement

À PROPOS La famille des méthodes de Forward

L'appel à `forward404Unless()` est équivalent à :

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

qui est similaire à :

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

La méthode `forward404()` elle-même est simplement un raccourci pour :

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

La méthode `forward()` redirige vers une autre action de la même application. Dans l'exemple précédent, il s'agit de l'action `404` du module `default`. Le module par défaut est livré avec Symfony et fournit les actions par défaut pour générer les pages 404, ainsi que les pages de contrôle d'accès et d'identification.

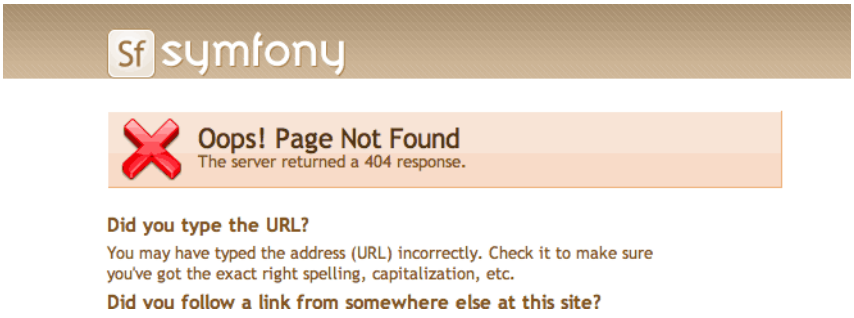


Figure 4–9
Page d'erreur 404 en environnement de production

La personnalisation de la page d'erreur 404 sera présentée plus loin dans cet ouvrage lorsqu'il sera temps de déployer l'application sur le serveur de production.

Comprendre l'interaction client/serveur

Lorsque l'on navigue sur les pages /job ou /job/show/id/1 dans un navigateur, un aller-retour avec le serveur web est effectué. Le navigateur envoie une requête et le serveur lui retourne une réponse.

Nous avons déjà vu que Symfony encapsulait la requête dans un objet `sfWebRequest` (vu dans la signature de la méthode `executeShow()`). Le framework étant entièrement orienté objet, la réponse est elle aussi un objet de la classe `sfWebResponse`. L'accès à l'objet réponse depuis une action se réalise au moyen de l'instruction `$this->getResponse()`.

Ces objets fournissent un certain nombre de méthodes adéquates pour accéder aux informations issues des fonctions et des variables globales de PHP.

CHOIX DE CONCEPTION Pourquoi Symfony encapsule-t-il des fonctionnalités existantes de PHP ?

Tout d'abord parce que les méthodes du framework sont plus puissantes que leurs équivalents en PHP. Ensuite, lorsque l'on teste une application, il devient beaucoup plus simple de simuler un objet requête ou un objet réponse que d'essayer de bricoler un programme avec des variables globales ou des fonctions PHP comme `header()`, dont le comportement est occulté par le développeur.

Récupérer le détail de la requête envoyée au serveur

La classe `sfWebRequest` encapsule les tableaux globaux `$_SERVER`, `$_COOKIE`, `$_GET`, `$_POST`, et `$_FILES` de PHP.

Tableau 4–1 Liste des méthodes disponibles de l'objet sfWebRequest

Nom de la méthode	Equivalent 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']`

Tableau 4–1 Liste des méthodes disponibles de l'objet sfWebRequest (suite)

Nom de la méthode	Equivalent PHP
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']

Les paramètres de la requête ont déjà été accédés en utilisant la méthode `getParameter()`. Celle-ci retourne une valeur en provenance de la variable globale `$_GET`, `$_POST`, ou bien de la variable `PATH_INFO`. Si l'on souhaite s'assurer qu'un paramètre provient bien d'une de ces variables, il faut alors avoir recours respectivement à l'une des méthodes `getGetParameter()`, `getPostParameter()` et `getUrlParameter()`.

Le framework Symfony introduit aussi la méthode `isMethod()` qui permet de contrôler la méthode HTTP utilisée ou de servir à restreindre une action à une méthode spécifique. C'est exactement ce qu'il se passe lorsque l'on manipule des formulaires. Grâce à la méthode `isMethod()`, on peut ainsi s'assurer directement dans l'action que la requête a été correctement transmise à l'aide de la méthode `POST` : `$this->forwardUnless($request->isMethod('POST'));`

Récupérer le détail de la réponse envoyée au client

La classe `sfWebResponse()` encapsule les fonctions PHP `header()` et `setrawcookie()`.

Tableau 4–2 Liste des méthodes disponibles de l'objet sfWebResponse

Nom de la méthode	Equivalent PHP
setCookie()	setrawcookie()
setStatusCode()	header()
setHttpHeader()	header()
setContentType()	header()
addVaryHttpHeader()	header()
addCacheControlHttpHeader()	header()

Bien sûr, la classe `sfWebResponse` fournit également une manière de fixer le contenu de la réponse (`setContent()`) et d'envoyer cette dernière au navigateur (`send()`).

Plus haut dans ce chapitre, nous avons découvert comment gérer les feuilles de style et les JavaScripts aussi bien dans le fichier `view.yml` que dans les templates. Au final, les deux techniques s'appuient sur les méthodes `addStyleSheet()` et `addJavascript()` de l'objet réponse.

Les classes `sfAction`, `sfRequest` et `sfReponse` fournissent un lot d'autres méthodes pratiques. N'hésitez pas à parcourir la documentation de l'API (http://www.symfony-project.org/api/1_2/) pour en savoir plus au sujet des classes internes de Symfony.

En résumé…

Ce chapitre a permis de présenter deux motifs de conception intégrés dans Symfony pour répondre aux besoins d'organisation du code. L'arborescence des fichiers du projet prend maintenant tout son sens puisque chaque chose est idéalement rangée à sa place. Dans la foulée, nous en avons profité pour apprivoiser la vue en manipulant le layout et les fichiers de templates. Certains d'entre eux ont d'ailleurs été rendus dynamiques par l'intermédiaire des slots et des actions.

Le chapitre suivant présente le sous-framework de routage. Ce sera ainsi l'occasion d'en savoir un peu plus à propos du helper `url_for()` qui a été vaguement aperçu et mis en œuvre au cours de ce chapitre.

chapitre 5

```
********************************
*                              *
*  symfony requirements check  *
*                              *
********************************

php.ini used by PHP: /apache2/php/etc/php.ini


** Mandatory requirements **

  OK        requires PHP >= 5.2.4
  OK        php.ini: requires zend.ze1_compatibility_mode set to off

** Optional checks **

  OK        PDO is installed
  OK        PDO has some drivers installed: sqlite2, sqlite, mysql
  OK        PHP-XML module installed
[[WARNING]] XSL module installed
            *** Install the XSL module (recommended for Propel) ***
  OK        can use token_get_all()
  OK        can use mb_strlen()
  OK        can use iconv()
  OK        can use utf8_decode()
  OK        has a PHP accelerator
  OK        php.ini: short_open_tag set to off
  OK        php.ini: magic_quotes_gpc set to off
  OK        php.ini: register_globals set to off
  OK        php.ini: session.auto_start set to off
```

Le routage

MOTS-CLÉS :

- Routage
- URL propres
- Objets sfRoute et sfDoctrineRoute

Toute ressource disponible sur un site web est identifiée au moyen d'une adresse Internet unique qui permet de l'atteindre. Ces URL jouent un rôle déterminant dans la sémantique et le référencement du site car elles apportent des informations utiles sur la ressource identifiée, et c'est pour cette raison qu'il est important de réfléchir à la manière dont elles seront implémentées.

Le framework Symfony intègre parfaitement un mécanisme interne puissant de gestion des « URLs propres ».

Le chapitre précédent a permis de découvrir et de se familiariser avec l'architecture MVC qui devient avec la pratique une manière de coder de plus en plus naturelle. C'est en s'exerçant davantage avec cette méthode de conception que l'on s'aperçoit à quel point il est délicat de s'organiser autrement… Le chapitre 3 a également permis de s'entraîner un peu plus en personnalisant certaines pages de Jobeet, mais aussi de revoir plusieurs concepts importants de Symfony comme le layout, les helpers ou encore les slots. À présent, il est temps de s'intéresser à un autre outil indispensable de Symfony : le framework de routage.

À la découverte du framework de routage de Symfony

Rappels sur la notion d'URL

En cliquant sur la page détaillée d'une offre d'emploi, l'URL ressemble à `/jobs/show/id/1`. Pourtant, les développeurs d'applications web PHP traditionnelles sont généralement plus familiers des URLs paramétrées telles que `/job.php?id=1`. Comment Symfony est-il capable de se comporter de la sorte avec les URLs ? Comment le framework détermine-t-il l'action à exécuter d'après cette URL ? Pourquoi l'identifiant d'une offre d'emploi est-il récupéré avec `$request->getParameter('id')` ? Ce sont toutes les questions auxquelles répond ce cinquième chapitre. Pour commencer, il est important de rappeler ce qu'est une URL dans le contexte du Web en général, et quel rôle elle joue exactement.

Qu'est-ce qu'une URL ?

Dans le contexte du Web, une URL est l'identifiant unique d'une ressource accessible depuis un navigateur web (une page HTML, une image, un fichier texte, une vidéo…). Schématiquement, lorsqu'un utilisateur saisit une adresse Internet dans son navigateur, il demande à ce dernier de lui récupérer la ressource distante identifiée par cette URL. Par conséquent, une URL se comporte comme une interface entre le site Internet et l'utilisateur, et peut ainsi véhiculer des informations utiles au sujet de la ressource qu'elle référence.

Or, un problème se pose avec les URLs paramétrées « traditionnelles » puisqu'elles ne décrivent pas véritablement la ressource distante et exposent, par la même occasion, l'implémentation technique interne de l'application. L'utilisateur se moque éperdument de savoir que le site Internet qu'il consulte est développé avec le langage PHP, ou bien que

chaque offre d'emploi est identifiée par un numéro unique dans la base de données. La seule chose qui l'intéresse, c'est d'accéder à la ressource qu'il désire grâce à cette URL.

Exposer les implémentations techniques internes de l'application se révèle par ailleurs particulièrement dangereux pour la sécurité de celle-ci. En effet, quels seraient les éventuels dégâts provoqués si un utilisateur malveillant arrivait à deviner l'URL de ressources auxquelles il n'a pas le droit d'accéder ? Bien évidemment, c'est le rôle du développeur de sécuriser de la meilleure manière possible son application, mais il reste toujours préférable de limiter les risques en cachant les informations sensibles.

Introduction générale au framework interne de routage

Au final, les adresses Internet sont si importantes dans Symfony qu'un framework entier leur est consacré : le framework de routage. Le système de routage gère à la fois les URLs internes et externes. Lorsqu'une requête entrante parvient au contrôleur frontal de l'application, celui-ci délègue le travail d'analyse de l'URL au framework interne de routage qui se charge de la convertir en une requête interne comme celle de la page d'une offre d'emploi, dans le template `showSuccess.php`.

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

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

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

Une URL interne par défaut est composée de plusieurs parties : tout d'abord le module `job`, suivi de l'action `show` et enfin de la chaîne de requête qui contient les paramètres à passer à l'action. Le motif générique d'une URL interne suit le format suivant :

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

Le framework de routage de Symfony est bidirectionnel, ce qui signifie que toutes les URLs peuvent être modifiées sans avoir à changer leur implémentation technique. C'est l'un des principaux avantages du motif de conception *Front Controller*.

Configuration du routage : le fichier routing.yml

Découverte de la configuration par défaut du routage

La configuration de toutes les URLs d'une application Symfony se situe dans un seul et même fichier de configuration YAML : le fichier

routing.yml. Celui-ci permet en effet de définir toute la carte des URLs internes de l'application de manière très simple. C'est ce qui fait aussi que le framework de routage est bidirectionnel puisque la modification de la configuration d'une URL dans ce fichier n'impactera pas son implémentation technique dans les templates ou dans les actions du projet. Le code ci-dessous décrit le contenu par défaut de ce fichier. Il s'agit de la déclaration des routes des trois motifs d'URLs nécessaires au fonctionnement de base du framework.

Configuration par défaut du routage dans le fichier 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 toutes les routes de l'application à l'aide d'une syntaxe YAML particulièrement simple. En effet, une route se déclare au minimum avec trois paramètres. Le premier d'entre eux est tout d'abord le nom donné à la route (homepage) afin d'y faire référence dans les actions et les templates lors de son implémentation technique.

Le deuxième paramètre (url) détermine bien évidemment le motif que doit prendre l'URL dans son implémentation finale. Par exemple, le motif /:module/:action/* indique que l'adresse Internet est composée d'une barre oblique, suivie d'une valeur pour la variable module, elle même suivie d'une barre oblique et d'une valeur pour la variable action, et enfin d'une dernière barre oblique suivie d'un caractère étoile qui indique au framework de routage que la route accepte une série de paramètres supplémentaires facultatifs.

Enfin, le dernier paramètre (param) est un tableau associatif dans lequel sont déclarées les valeurs par défaut de certaines variables propres à l'URL. Par exemple, la route homepage force le framework de routage à exécuter l'action index du module default en guise de page d'accueil. Les sections qui suivent abordent plus en détail tous ces aspects de configuration des routes de l'application.

Comprendre le fonctionnement des URL par défaut de Symfony

Lorsqu'une requête arrive sur le contrôleur frontal, le système de routage tente de lui faire correspondre un motif d'URL. La première route identifiée l'emporte, ce qui signifie que l'ordre de déclaration des routes dans le fichier `routing.yml` est important. Voici quelques exemples pour mieux comprendre comment cela fonctionne.

Quand un utilisateur demande la page d'accueil des offres d'emploi, dont l'URL est `/job`, la première route qui correspond à ce motif est en effet `default_index`. Dans un motif, un mot préfixé par deux points `:` est en fait une variable. Par conséquent, le motif `/:module` signifie : « trouve un `/` suivi par quelque chose qui sera ensuite stocké dans la variable `module` ». Dans cet exemple, la variable `module` prend pour valeur `job` et peut ensuite être retrouvée dans l'action grâce à l'instruction `$request->getParameter('module')`. Cette route définit aussi une valeur par défaut pour la variable `action`. Ainsi, pour toutes les URLs identifiables avec cette route, la requête disposera d'un paramètre `action` dont la valeur est `index`.

Si la page `/job/show/id/1` est demandée, Symfony fera correspondre son URL au dernier motif : `/:module/:action/*`. Dans un motif, l'étoile `*` indique une collection de paires variable/valeur séparées par des barres obliques `/`.

Tableau 5–1 Liste des paramètres et valeurs de l'URL de la page d'une offre d'emploi

Paramètre	Valeur
`module`	`job`
`action`	`show`
`id`	`1`

L'URL `/job/show/id/1` peut être créée depuis un template en ayant recours à l'appel au helper `url_for()` suivant :

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

Le même résultat est également possible à partir du nom de la route préfixé par un arobase `@` :

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

Les deux appels sont équivalents, mais le dernier est nettement plus performant. En effet, le framework de routage n'a pas besoin d'analyser chaque route pour déterminer celle qui correspond le mieux. De plus, il est moins lié à l'implémentation du nom du module et de l'action puisque leur valeur respective est absente de l'URL interne.

REMARQUE

Les variables spéciales module et action

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

Personnaliser les routes de l'application

Configurer la route de la page d'accueil

Pour l'instant, la page d'accueil de l'application Jobeet est toujours celle par défaut de félicitations de Symfony. C'est en effet parce que l'adresse Internet interne `/` correspond à la route `homepage` définit dans le fichier de configuration `routing.yml`. Il faut donc remplacer cette page d'accueil par défaut au profit de celle de Jobeet. Pour ce faire, il suffit d'éditer la configuration initiale de la route `homepage` en remplaçant la valeur de la variable `module` par `job`.

Configuration de la page d'accueil de Jobeet dans le fichier apps/frontend/config/routing.yml

```
homepage:
  url:   /
  param: { module: job, action: index }
```

Ceci étant fait, le lien figurant sur le logo de chaque page de Jobeet peut également être édité afin d'appliquer l'URL vers la page d'accueil de l'application.

Définition du lien vers la page d'accueil dans le fichier apps/frontend/templates/layout.php

```
<h1>
  <a href="<?php echo url_for('@homepage') ?>">
    <img src="/images/jobeet.gif" alt="Jobeet Job Board" />
  </a>
</h1>
```

Configurer la route d'accès au détail d'une offre

La modification de la configuration d'une URL n'implique que de changer certains paramètres dans le fichier de configuration `routing.yml`. L'objectif à présent est d'aller de l'avant en transformant le motif de l'URL qui mène au détail d'une annonce, afin que cette adresse embarque davantage d'informations utiles comme le nom de la société, la ville ou bien encore le type de poste proposé. Cela permet à la fois de deviner par avance à quoi s'attendre en atteignant cette URL, mais aussi d'optimiser le référencement du site auprès des moteurs de recherche qui indexent les mots-clés qu'ils trouvent dans les adresses Internet. Le nouveau motif des URLs de chaque offre correspondra à celui ci-dessous :

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

Grâce à ce motif, l'utilisateur ne connaissant absolument rien de Jobeet est capable de comprendre que la société parisienne Sensio Labs est à la recherche d'un nouveau développeur web. Cette URL est certes plus longue que celle par défaut mais elle a l'avantage d'être bien plus pertinente et sémantique.

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

Pour y parvenir, il faut bien entendu modifier le contenu du fichier de configuration `routing.yml` afin de lui ajouter une route supplémentaire `job_show_user` dont la configuration est donnée dans le code ci-après. Le motif de cette route fait état de quatre variables séparées les unes des autres par des barres obliques. Les variables `company`, `location`, `id` et `position` représentent respectivement le nom de la société, le lieu, l'identifiant unique et le type de poste proposé pour l'offre courante.

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

En rafraîchissant de nouveau la page d'accueil, les liens vers les pages respectives des offres d'emploi n'ont pas changé. C'est en effet normal étant donné que la route accepte à présent des paramètres obligatoires qui doivent être transmis au helper `url_for()` afin qu'il génère l'URL adéquate. Par conséquent, la définition du helper `url_for()` dans le template `indexSuccess.php` du module `job` doit être modifiée de la manière suivante.

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

Il est également possible d'exprimer une URL interne à l'aide d'un tableau associatif.

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

REMARQUE De l'utilité des URLs propres

Les URLs propres et bien formées sont importantes car elles véhiculent des informations à l'utilisateur. C'est aussi particulièrement pratique lorsque l'on copie/colle l'URL dans un e-mail ou lorsqu'il s'agit d'optimiser un site Internet pour les moteurs de recherche qui se servent des mots-clés présents dans les URLs.

Forcer la validation des paramètres des URLs internes de l'application

Pour des raisons évidentes de sécurité et d'aide au débogage, le premier chapitre a mis en évidence les notions de validation et de gestion des erreurs. Le système de routage n'échappe pas non plus à cette règle puisqu'il possède une fonctionnalité native de validation des paramètres des URLs internes. La valeur de chaque variable d'une adresse interne ayant un format propre peut être validée au moyen d'une expression régulière, définie par l'intermédiaire de la section `requirements` de la configuration d'une route.

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

La section `requirements` ci-dessus force la valeur de la variable `id` à être une valeur numérique entière strictement positive. Si ce n'est pas le cas, la route ne correspondra pas au motif.

Limiter une requête à certaines méthodes HTTP

Chaque route configurée dans le fichier `routing.yml` est convertie en interne sous la forme d'un objet de la classe `sfRoute`. Il arrive parfois qu'il faille écrire des routes plus complexes se comportant différemment des routes traditionnelles. Par conséquent, la classe `sfRoute` doit être remplacée par une classe plus spécifique.

L'entrée `class` de la configuration des routes du fichier `routing.yml` permet au développeur de modifier le nom de la classe à utiliser pour contrôler les comportements de l'URL interne courante. Par exemple, le lecteur familier du protocole HTTP sait que celui-ci accepte les méthodes `GET`, `POST`, `HEAD`, `DELETE` ou encore `PUT`, bien que les navigateurs web ne supportent que les trois premières. Par conséquent, il mesure tout l'intérêt de pouvoir limiter l'utilisation d'une requête HTTP pour une ou plusieurs de ces méthodes.

En remplaçant la classe `sfRoute` par la classe `sfRequestRoute` et en ajoutant une contrainte à la variable virtuelle `sf_method`, le framework interne de routage *force* la route à n'employer que certaines méthodes de requêtes HTTP. Comme les méthodes `HEAD` et `PUT` ne sont pas supportées par les navigateurs web modernes, le framework Symfony est

REMARQUE La méthode sfWebRequest::isMethod() vs framework de routage

Forcer une route à correspondre à certaines requêtes HTTP n'est pas entièrement équivalent à utiliser `sfWebRequest::isMethod()` dans les actions. En effet, dans le premier cas, le routage continuera de chercher une route correspondante si la méthode ne correspond pas à celle attendue.

capable d'émuler leur comportement grâce notamment à l'utilisation de cette variable spéciale `sf_method`.

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

Optimiser la création de routes grâce à la classe de route d'objets Doctrine

La nouvelle URL interne pour les offres d'emploi est particulièrement longue et fastidieuse à écrire puisqu'il est nécessaire de passer l'intégralité des paramètres obligatoires à la route par le biais du helper `url_for()`. Hormis le fait qu'elle soit contraignante à écrire, elle dispose d'un second inconvénient tout aussi ennuyeux. En effet, l'implémentation technique de la route et sa déclaration dans le fichier de configuration `routing.yml` sont fortement couplées, ce qui implique que des modifications dans le paramétrage de l'URL affecteront aussi l'implémentation technique dans les templates et les actions. Par exemple, si un nouveau paramètre est ajouté au motif de l'URL, alors il faudra penser à modifier tous les templates et les actions pour passer la valeur de ce dernier. Ce n'est ni pratique ni intéressant pour gagner du temps.

Transformer la route d'une offre en route Doctrine

L'idéal est donc de se tourner vers une approche différente de la création d'URLs complexes. La section précédente a montré comment il était simple de modifier la manière dont est gérée une route en modifiant le nom de sa classe associée. De ce fait, il convient d'avoir recours à la classe `sfDoctrineRoute` pour manipuler la route `job_show_user`, dans la mesure où cette classe est optimisée pour représenter n'importe quel objet Doctrine ou n'importe quelle collection 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]
```

La section `options` personnalise le comportement de la route. Ici, l'option `model` indique la classe de modèle (`JobeetJob`) relative à la route, tandis que l'option `type` précise que la route est liée à un objet. Si la valeur de l'option `type` est `list` alors la route se rapportera à une collection d'objets.

La route `job_show_user` est maintenant informée de sa relation avec la classe de modèle `JobeetJob`, ce qui permet de simplifier l'appel à `url_for()` par :

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

Ou encore tout simplement :

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

ASTUCE **Choisir le bon appel à url_for()**

Le premier des deux appels à `url_for()` présentés ici est plus pratique lorsqu'il s'agit de transmettre des paramètres supplémentaires autres que l'objet relatif à la route Doctrine lui-même.

Améliorer le format des URL des offres d'emploi

L'implémentation de la route Doctrine précédente fonctionne telle quelle car toutes les variables qui se trouvent dans le motif de l'URL disposent en réalité d'un accesseur correspondant dans la classe `JobeetJob`. Par exemple, la valeur de la variable `company` est rendue automatiquement à l'appel implicite à la méthode `getCompany()`. Toutefois, les URLs générées pour certaines offres ne sont pas véritablement celles désirées puisque en effet certaines données comme la localisation ou bien le type de poste contiennent des caractères non standards pour une URL.

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

REMARQUE **Qu'est-ce que la création de slugs ?**

L'action de réalisation d'un slug à partir d'une chaîne de départ consiste à transformer cette dernière en vue de la réutiliser dans une URL comme identifiant d'une ressource ou seulement comme mots-clés supplémentaires pour véhiculer de l'information au sujet de la ressource identifiée. Cette technique apporte de nombreux avantages pour une URL car elle permet à la fois de rendre cette dernière plus claire, plus lisible mais surtout plus sémantique pour l'utilisateur. D'autre part, les mots-clés qu'elle véhicule participent à l'optimisation de l'indexation du site Internet auprès des moteurs de recherche.

Il apparaît donc nécessaire de transformer à la volée les valeurs de ces colonnes en remplaçant tous les caractères non ASCII par des tirets - afin d'obtenir ce qu'on appelle des « slugs » dans le jargon informatique. N'ayant pas de véritable traduction courte en français, le terme *slug* sera employé tout au long de cet ouvrage pour désigner une chaîne optimisée pour une URL. Pour ce faire, de nouvelles méthodes doivent être ajoutées à la classe `JobeetJob`.

Méthodes à ajouter à la classe JobeetJob dans le fichier 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());
}
```

Ces trois méthodes font appel à une nouvelle classe `Jobeet` dans laquelle se trouve une méthode statique `slugify()`. C'est cette méthode qui se charge de transformer une chaîne de caractères d'origine sous la forme d'une chaîne simplifiée et optimisée pour une URL. Il convient donc de créer le fichier `lib/Jobeet.class.php` dans lequel est implémentée cette nouvelle classe `Jobeet`.

Implémentation de la classe Jobeet dans le fichier lib/Jobeet.class.php

```
<?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;
  }
}
```

La dernière étape consiste à remanier la définition de la route `job_show_user` afin que celle-ci fasse désormais usage des trois nouveaux accesseurs virtuels de la classe `JobeetJob` à la place des trois accesseurs actuels. La modification de la route consiste en fait uniquement à changer les noms des paramètres dans le motif de l'URL sans avoir à modifier quoi que ce soit dans le template.

Édition de la route job_show_user dans le fichier de configuration apps/frontend/config/routing.yml

```
job_show_user:
  url:     /job/:company_slug/:location_slug/:id/:position_slug
  class:   sfDoctrineRoute
  options: { model: JobeetJob, type: object }
  param:   { module: job, action: show }
```

REMARQUE

Suppression des balises PHP <?php

Tout au long de cet ouvrage, les balises d'ouverture de script PHP `<?php` seront volontairement omises dans les morceaux de code présentés. C'est en fait tout simplement par économie de place et pour rendre le code plus agréable à lire en retirant le « bruit » que génèrent ces balises. Toutefois, elles sont obligatoires à l'exécution des scripts PHP par le serveur. Il faut donc prendre garde à ne pas les oublier lors de la réimplémentation des codes présentés.

```
  requirements:
    id: \d+
    sf_method: [get]
```

Il ne reste plus qu'à vider le cache de Symfony (commande `symfony cc`), étant donné qu'une nouvelle classe `Jobeet` a été ajoutée au projet, afin de pouvoir constater la transformation des chaînes de caractères passées dans les URLs des offres d'emploi.

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

Retrouver l'objet grâce à sa route depuis une action

La puissance du framework de routage se trouve encore au-delà des concepts étudiés jusqu'à présent. En effet, la route est capable de générer une URL basée sur un objet, mais aussi de récupérer celui-ci grâce à l'objet `sfDoctrineRoute`. L'objet lié peut alors être retrouvé en utilisant la méthode `getObject()` de l'objet de la route Doctrine. Lorsque le système de routage analyse une requête entrante, il garde en mémoire l'objet de la route correspondante afin de l'utiliser dans les actions. Ainsi, la méthode `executeShow()` est désormais en mesure de retrouver l'objet `JobeetJob` grâce à l'objet de la route Doctrine.

Extrait de la méthode executeShow() dans le fichier apps/frontend/modules/job/actions/actions.class.php

```
class jobActions extends sfActions
{
  public function executeShow(sfWebRequest $request)
  {
    $this->job = $this->getRoute()->getObject();
    $this->forward404Unless($this->job);
  }

  // ...
}
```

L'appel à la méthode `getObject()` de l'objet de la route Doctrine lance une exception si l'objet Doctrine lié n'existe pas, ce qui provoque une page d'erreur 404 dont le message d'erreur est différent de celui renvoyé habituellement en environnement de développement comme le montre la capture d'écran un peu plus bas. Par conséquent, le corps de la méthode `executeShow()` peut être simplifié puisqu'il n'est plus nécessaire de gérer soi-même la redirection vers une page d'erreur 404 lorsque aucun objet n'est renvoyé.

Simplification de la méthode executeShow() dans le fichier apps/frontend/modules/job/actions/actions.class.php

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

  // ...
}
```

Figure 5–1 Page d'erreur 404 d'une route Doctrine en environnement de développement

Faire appel au routage depuis les actions et les templates

Le routage dans les templates

Dans un template, le helper `url_for()` convertit une URL interne en une URL externe. D'autres helpers Symfony prennent aussi une URL interne comme argument, comme le helper `link_to()` qui génère une balise HTML `<a>`.

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

La portion de code ci-dessus génère le code HTML suivant :

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

Les deux helpers `url_for()` et `link_to()` peuvent produire des URLs absolues si on leur fournit un nouveau paramètre booléen facultatif.

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

ASTUCE Désactiver la levée des exceptions de la méthode getObject()

Pour éviter que la méthode `getObject()` ne lève une erreur 404, l'option `allow_empty` peut être définie à la valeur `true` dans la configuration de la route.

REMARQUE Chargement à la demande de l'objet Doctrine d'une route

L'objet lié à la route est chargé à la demande, c'est-à-dire qu'il est récupéré de la base de données uniquement lorsque la méthode `getRoute()` est appelée.

Le routage dans les actions

Le routage trouve aussi sa place dans les actions lorsqu'il s'agit par exemple d'effectuer une redirection automatique vers une page de l'application après que l'utilisateur a réalisé une opération, comme le remplissage d'un formulaire. Les redirections sont rendues possibles depuis les actions par le biais de la méthode `redirect()` tandis que la génération des URLs spécifiques est réalisée grâce à la méthode `generateUrl()`.

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

> ASTUCE **La famille des méthodes de redirection**
>
> Dans le précédent chapitre, il était question des méthodes de *forward*. Ces méthodes transmettent la requête en cours à une autre action sans réaliser d'échange avec le serveur.
> Les méthodes de *redirect* redirigent l'utilisateur vers une autre URL. Comme pour *forward*, il est possible d'utiliser `redirect()`, ou ses raccourcis `redirectIf()` et `redirectUnless()`.

Découvrir la classe de collection de routes sfDoctrineRouteCollection

La route de l'action `show` du module d'offres d'emploi a déjà été personnalisée, mais les URLs des autres méthodes (`index`, `new`, `edit`, `create`, `updated` et `delete`) sont toujours gérées par la route par défaut.

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

La route par défaut est un moyen très pratique de commencer à coder sans définir trop de routes. Mais dès lors que la route agit comme un *catch-all* (littéralement *fourre-tout* en français), elle ne peut plus être configurée pour des besoins spécifiques.

Déclarer une nouvelle collection de routes Doctrine

Comme toutes les actions d'une offre d'emploi sont relatives à la classe de modèle `JobeetJob`, il est possible de définir une route `sfDoctrineRoute` personnalisée pour chacune d'elles au même titre que celle mise en œuvre avec l'action `show`. Or, le module d'offres d'emploi définit sept actions de base possibles pour le modèle. Il est donc préférable d'avoir recours à la classe `sfDoctrineRouteCollection`. L'utilisation de cette classe nécessite de modifier le fichier `routing.yml` comme suit.

Configuration d'une collection de routes pour l'objet JobeetJob dans le fichier 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 n'est qu'un simple 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
```

REMARQUE Routes identiques dans une collection de routes Doctrine

Certaines routes générées par la classe `sfDoctrineRouteCollection` ont la même URL. Le framework de routage est en fait capable de les utiliser car elles possèdent toutes différentes méthodes HTTP obligatoires.

```
  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 }
```

Émuler les méthodes PUT et DELETE

Les routes `job_delete` et `job_update` nécessitent l'utilisation des méthodes HTTP `DELETE` et `PUT` qui ne sont pas supportées par les navigateurs web. Néanmoins, ces URLs fonctionnent quand même étant donné que Symfony arrive à les simuler à l'aide de la variable spéciale `sf_method`. Le template `_form.php` donne un exemple d'implémentation de ce mécanisme.

Extrait du fichier 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 de Symfony sont capables d'émuler n'importe quelle méthode HTTP lorqu'on leur passe le paramètre spécial `sf_method`. Le framework possède d'autres paramètres particuliers comme `sf_method`, qui débutent tous par le préfixe `sf_`. Les routes générées plus haut dans

cette section possèdent toutes un – `sf_format` – qui sera présenté à l'occasion d'un chapitre consacré aux services web. Le chapitre dédié à l'internationalisation et la localisation de Jobeet fera quant à lui usage de la variable spéciale `sf_culture`.

Outils et bonnes pratiques liés au routage

Faciliter le débogage en listant les routes de l'application

Plus l'application développée grandit et plus le nombre de routes déclarées croît, ce qui signifie aussi qu'il devient de plus en plus difficile de s'y retrouver entre les différentes URLs. C'est d'autant plus vrai lorsque l'application connecte des collections de routes Doctrine. En effet, les collections de routes Doctrine sont définies globalement dans le fichier de configuration `routing.yml`, mais la configuration des routes Doctrine unitaires qu'elles renferment n'est pas visible au travers de ce fichier. Par conséquent, le débogage d'une route particulière de l'application peut se révéler plus complexe. Heureusement, le framework Symfony intègre une tâche automatique `app:routes` qui permet de connaître l'intégralité des routes connectées à l'application comme en témoigne le résultat ci-après.

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

L'exécution de cette commande produit un résultat comparable à celui en dessous. Cette liste donne pour chaque route connectée son nom, la méthode qui restreint son accès, et bien sûr son motif complet qui tient compte de la variable spéciale `sf_format` présentée plus tard.

```
>> app       Current routes for application "frontend"
Name               Method Pattern
job             GET    /job.:sf_format
job_new         GET    /job/new.:sf_format
job_create      POST   /job.:sf_format
job_edit        GET    /job/:id/edit.:sf_format
job_update      PUT    /job/:id.:sf_format
job_delete      DELETE /job/:id.:sf_format
job_show        GET    /job/:id.:sf_format
job_show_user GET    /job/:company_slug/:location_slug/:id/
:position_slug
homepage        ANY    /
```

Il est également possible d'obtenir de nombreuses informations de débogage pour une route donnée en passant son nom en tant que second paramètre additionnel.

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

L'exécution de cette commande a pour effet de produire le résultat suivant dans le terminal. Cette commande est particulièrement intéressante puisqu'elle donne tout le détail de la configuration de la route, y compris les paramètres internes définis automatiquement par la classe `sfDoctrineRoute`. Les éléments mis en exergue sont ceux qui ont déjà été vus tout au long de ce chapitre.

```
>> app       Route "job_edit" for application "frontend"
Name         job_edit
Pattern      /job/:id/edit.:sf_format
Class        sfDoctrineRoute
Defaults     action: 'edit'
             module: 'job'
             sf_format: 'html'
Requirements id: '\\d+'
             sf_format: '[^/\\.]+'
             sf_method: 'get'
Options      context: array ()
             debug: false
             extra_parameters_as_query_string: true
             generate_shortest_url: true
             load_configuration: false
             logging: false
             method: NULL
             model: 'JobeetJob'
             object_model: 'JobeetJob'
             segment_separators: array (0 => '/',1 => '.',)
             segment_separators_regex: '(?:/|\\.)'
             suffix: ''
             text_regex: '.+?'
             type: 'object'
             variable_content_regex: '[^/\\.]+'
             variable_prefix_regex: '(?:\\:)'
             variable_prefixes: array (0 => ':',)
             variable_regex: '[\\w\\d_]+'
Regex        #^
             /job
             /(?P<id>\d+)
             /edit
             (?:\.(?P<sf_format>[^/\.]+)
             )?
             $#x
Tokens       separator array (0 => '/',1 => NULL,)
             text       array (0 => 'job',1 => NULL,)
             separator array (0 => '/',1 => NULL,)
             variable   array (0 => ':id',1 => 'id',)
             separator array (0 => '/',1 => NULL,)
             text       array (0 => 'edit',1 => NULL,)
             separator array (0 => '.',1 => NULL,)
             variable   array (0 => ':sf_format',1 => 'sf_format',)
```

Supprimer les routes par défaut

Une bonne pratique consiste à déclarer une route spécifique pour chaque URL de l'application, et à supprimer les routes par défaut afin d'empêcher l'accès à des ressources pour lesquelles aucune route dédiée n'aurait été déclarée dans le fichier de configuration `routing.yml`. Cela permet par exemple de se prémunir des pirates qui tenteraient d'accéder à des modules de l'application non sécurisés en passant par les routes par défaut qui correspondent à n'importe quel motif d'URL.

Par conséquent, comme la route `job` définit toutes les routes nécessaires pour décrire l'application Jobeet, les routes par défaut du fichier de configuration `routing.yml` peuvent être supprimées ou mises en commentaire en toute sécurité. L'application Jobeet devrait continuer de fonctionner normalement comme avant.

Désactivation des routes par défaut de l'application frontend dans le fichier apps/frontend/config/routing.yml

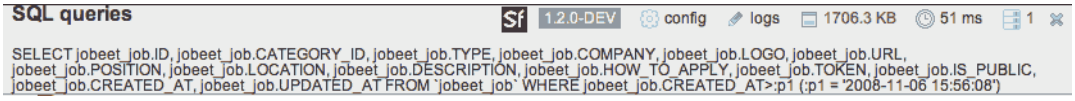
```
#default_index:
# url:   /:module
# param: { action: index }
#
#default:
# url:   /:module/:action/*
```

En résumé…

Ce chapitre a présenté de nombreux concepts importants concernant le framework interne de routage de Symfony comme les routes par défaut, basiques, restreintes aux méthodes HTTP, d'objets et de collections Doctrine, ou encore les collections de routes Doctrine. De plus, ces pages ont montré comment il est possible de créer des URLs élégantes et significatives avec Symfony, tout en les découplant de leur implémentation technique. Tous ces aspects seront très régulièrement remis en œuvre tout au long de cet ouvrage dans la mesure où les URLs jouent un rôle fondamental dans une application web.

Le chapitre suivant quant à lui n'introduit pas véritablement de nouveaux concepts, ce qui permettra de revenir plus en détail sur une bonne partie des notions abordées jusqu'à présent : le routage, l'architecture MVC, les objets Doctrine, le remaniement du code, etc.

chapitre 6


SQL queries
1.2.0-DEV
config
logs
1706.3 KB
51 ms
1
SELECT jobeet_job.ID, jobeet_job.CATEGORY_ID, jobeet_job.TYPE, jobeet_job.COMPANY, jobeet_job.LOGO, jobeet_job.URL, jobeet_job.POSITION, jobeet_job.LOCATION, jobeet_job.DESCRIPTION, jobeet_job.HOW_TO_APPLY, jobeet_job.TOKEN, jobeet_job.IS_PUBLIC, jobeet_job.CREATED_AT, jobeet_job.UPDATED_AT FROM `jobeet_job` WHERE jobeet_job.CREATED_AT>:p1 (:p1 = '2008-11-06 15:56:08')

Optimisation du modèle et refactoring

La majeure partie du code métier d'une application MVC est conditionnée dans la couche du modèle. Or, le volume de code du modèle peut très vite devenir un véritable casse-tête à maintenir et à pérenniser. C'est pourquoi ce chapitre sera l'occasion de vous familiariser avec les techniques de factorisation et de simplification du code.

MOTS-CLÉS :

- Modèle
- Doctrine Query Language
- Refactoring de code

Le chapitre précédent a permis d'aborder la manière de créer des URLs élégantes et de voir comment utiliser le framework Symfony pour automatiser de nombreuses choses. Dans les pages qui suivent, le site web Jobeet va être amélioré en optimisant le code ici et là. Dans la foulée, les fonctionnalités abordées jusqu'à maintenant seront un peu plus détaillées.

Présentation de l'objet Doctrine_Query

Parmi les objectifs définis au chapitre 2 figure celui-ci :

« Quand un utilisateur arrive sur le site web de Jobeet, il découvre une liste des offres d'emploi actives ».

Pour le moment, toutes les offres d'emploi sont affichées, qu'elles soient actives ou non.

Contenu du fichier apps/frontend/modules/job/actions/actions.class.php

```
class jobActions extends sfActions
{
  public function executeIndex(sfWebRequest $request)
  {
    $this->jobeet_job_list = Doctrine::getTable('JobeetJob')
      ->createQuery('a')
      ->execute();
  }

  // ...
}
```

Une offre active est une offre qui a été postée il y a moins de 30 jours. C'est la méthode `Doctrine_Query::execute()` qui effectue une requête sur la base de données. Dans le code ci-dessus, aucune condition particulière n'a été spécifiée, ce qui signifie que tous les enregistrements sont récupérés de la base de données.

Avec Doctrine, l'ajout de conditions dans une requête SQL est réalisé à l'aide de la méthode `where()`, comme le montre le code modifié de la méthode `executeIndex()` de la classe `JobActions` :

```
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_job_list = $q->execute();
}
```

Déboguer le code SQL généré par Doctrine

Dispensant d'avoir à écrire les requêtes SQL à la main, Doctrine fait attention aux différences entre les moteurs de bases de données, et génère les ordres SQL optimisés pour le moteur configuré au chapitre 3. Néanmoins, c'est parfois une aide indéniable que de pouvoir lire le code SQL généré par Doctrine ; par exemple, pour déboguer une requête qui ne fonctionne pas comme on l'attend. En environnement de développement, Symfony enregistre ces requêtes (et bien plus encore) dans le fichier `log/frontend_dev.log`.

Découvrir les fichiers de log

Le répertoire `log/` stocke les fichiers de log par application et par environnement, ce qui permet de retrouver et de suivre facilement l'ensemble des opérations internes effectuées par le framework lorsqu'une URL est demandée au serveur.

Exemple de log dans le fichier 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)
```

Il est ainsi possible de contrôler que la requête générée par Doctrine possède bel et bien une clause `Where` sur la colonne `created_at` (`WHERE j.created_at > ?`).

BONNE PRATIQUE Les requêtes préparées pour lutter contre les attaques par injections

La chaîne « ? » dans la requête indique que Doctrine génère des requêtes préparées. La valeur courante de « ? » (« 2008-11-08 01:13:35 » dans l'exemple ci-dessus) est passée au cours de l'exécution de la requête afin d'être traitée littéralement par le moteur de base de données. L'usage de requêtes préparées réduit drastiquement l'exposition de l'application aux attaques par injections SQL.

Avoir recours à la barre de débogage

Si l'accès à ces journaux d'événements reste très pratique, il ne demeure pas moins contraignant d'avoir à passer du navigateur à l'EDI (« environnement de développement intégré ») et au fichier de logs. C'est pourquoi Symfony comporte une barre de débogage afin de rendre disponibles depuis le navigateur toutes les informations pertinentes.

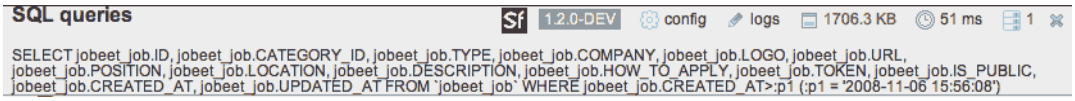


Figure 6–1 Requêtes SQL générées par Doctrine dans la barre d'outils de déboguage de Symfony

Intervenir sur les propriétés d'un objet avant sa sérialisation en base de données

Même si le code ci-dessus fonctionne, il est encore loin d'être parfait dans la mesure où il ne prend pas en compte certaines exigences telle que la suivante :

« Un utilisateur peut revenir pour réactiver ou étendre la validité de l'annonce d'une offre d'emploi pour une nouvelle période de 30 jours… »

Or, le code précédent s'appuie sur la valeur du champ `created_at`, et parce que cette colonne stocke la date de création, il est impossible de satisfaire ce besoin.

Redéfinir la méthode save() d'un objet Doctrine

Une colonne `expires_at` a heureusement été prévue dans le schéma de base de données (voir chapitre 3). Pour l'instant, sa valeur n'est pas renseignée puisqu'elle n'est pas définie dans le fichier de données initiales. Mais lors de la création d'une nouvelle offre d'emploi, cette colonne pourrait automatiquement prendre pour valeur la date courante du serveur, à laquelle une période de 30 jours est ajoutée.

Pour réaliser une opération juste avant qu'un objet Doctrine ne soit sérialisé en base de données, il suffit de redéfinir la méthode `save()` de la classe de modèle.

Redéfinition de la méthode save() de l'objet dans le fichier lib/model/doctrine/JobeetJob.class.php

```
class JobeetJob extends BaseJobeetJob
{
  public function save(Doctrine_Connection $conn = null)
  {
    if ($this->isNew() && !$this->getExpiresAt())
    {
      $now = $this->getCreatedAt() ?
              ➥ strtotime($this->getCreatedAt()) : time();
```

```
      $this->setExpiresAt(date('Y-m-d h:i:s',
                              $now + 86400 * 30));
    }

    return parent::save($conn);
  }

  // ...
}
```

La méthode `isNew()` retourne `true` quand l'objet n'a pas été sérialisé en base de données et `false` dans le cas contraire.

Récupérer la liste des offres d'emploi actives

Il est maintenant nécessaire de modifier l'action afin d'utiliser la colonne `expires_at` au lieu de `created_at` pour sélectionner les offres d'emploi actives.

```
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_job_list = $q->execute();
}
```

La requête est ainsi réduite à ne récupérer que les offres d'emploi ayant une valeur `expires_at` dans le futur.

Mettre à jour les données de test pour s'assurer de la validité des offres affichées

Rafraîchir la page d'accueil de Jobeet dans un navigateur ne changera rien puisque les offres d'emploi enregistrées en base de données n'ont été postées que quelques jours plus tôt. Il est donc obligatoire de changer les données de test pour ajouter une nouvelle offre d'emploi déjà expirée.

Offre d'emploi à ajouter au fichier data/fixtures/jobs.yml

```
JobeetJob:
  # other jobs

  expired_job:
    JobeetCategory: programming
    company:        Sensio Labs
```

IMPORTANT **Respecter l'indentation d'un fichier YAML**

Il faut toujours veiller à ne pas rompre l'indentation lorsque l'on copie et colle du code dans un fichier de données de tests. L'offre d'emploi `expired_job` ne doit avoir que deux espaces qui la précèdent.

```
  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
  expires_at:     '2005-12-01 00:00:00'
  token:          job_expired
  email:          job@example.com
```

Dans la définition des champs de cette offre nouvellement créée, la valeur de la colonne `created_at` a été redéfinie explicitement – bien qu'elle soit automatiquement remplie par Doctrine. La valeur ainsi spécifiée surchargera celle par défaut. En rechargeant les données de test, puis en rafraîchissant le navigateur, l'offre d'emploi expirée ne s'affichera pas.

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

La requête SQL suivante permet de contrôler que la colonne `expires_at` est bien remplie automatiquement par la méthode `save()` à partir de la valeur de `created_at`.

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

Gérer les paramètres personnalisés d'une application dans Symfony

Dans la méthode `JobeetJob::save()`, le nombre de jours restant avant qu'une offre d'emploi n'expire a été codé « en dur ». Or il vaudrait bien mieux rendre cette valeur configurable ailleurs – pour des raisons de facilité de maintenance et de généricité. La solution est toute trouvée avec le fichier de configuration `app.yml` qui permet de définir les paramètres spécifiques à une application. Ce fichier YAML, fourni par défaut par le framework Symfony, peut définir n'importe quel paramètre.

Découvrir le fichier de configuration app.yml

En règle générale, il est déconseillé de fixer en dur dans un programme des informations qui peuvent être configurées ailleurs. En effet, en centralisant les paramètres dans un fichier commun, les modifications se font plus simplement et plus rapidement – autant de temps gagné en maintenance.

Symfony fournit à cet usage un fichier dédié à la configuration d'une application. Il s'agit du fichier `apps/APPLICATION/config/app.yml`.

Exemple de fichier de configuration : contenu de apps/frontend/config/app.yml

```
all:
  active_days: 30
```

Dans l'application, ces paramètres sont disponibles au travers de la classe globale `sfConfig`.

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

Récupérer une valeur de configuration depuis le modèle

Le paramètre a été préfixé par `app_` parce que la classe `sfConfig` fournit aussi un accès aux paramètres de Symfony comme il sera présenté plus tard. Afin que ce nouveau paramètre soit pris en compte, il est nécessaire de modifier de nouveau la méthode `save()` de l'objet `JobeetJob`.

```
public function save(Doctrine_Connection $conn = null)
{
  if ($this->isNew() && !$this->getExpiresAt())
  {
    $now = $this->getCreatedAt() ?
           ➥ strtotime($this->getCreatedAt()) : 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 excellent moyen de centraliser les paramètres globaux de l'application. Ceux-ci restent alors disponibles à tout moment et depuis n'importe où grâce à la classe globale `sfConfig`.

Remanier le code en continu pour respecter la logique MVC

Le code écrit fonctionne parfaitement, mais il n'est pas encore tout à fait correct. Où se situe le problème et comment le résoudre ? Le chapitre 4 a montré comment le modèle MVC sépare le code en trois couches distinctes : le modèle, la vue et le contrôleur.

Tout au long du processus de développement de l'application, il sera souhaitable de garder cette règle à l'esprit afin de penser à remanier du code, voire le déplacer ailleurs lorsque nécessaire.

Exemple de déplacement du contrôleur vers le modèle

Le code de `Doctrine_Query` n'appartient pas à l'action (la couche contrôleur) ; il dépend en réalité de la couche du modèle. Dans le motif MVC, le modèle définit toute la logique métier, c'est-à-dire celle qui manipule les données, tandis que le contrôleur ne fait qu'appeler celui-ci pour y récupérer des données, en fonction de la requête de l'utilisateur, qu'il communique ensuite à la vue.

Dans le cas présent, le code retourne une collection d'offres d'emploi en guise de résultat ; c'est pourquoi il convient de déplacer ce dernier dans la classe `JobeetJobTable`, dans laquelle on crée une méthode `getActiveJobs()`.

Contenu du fichier 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();
  }
}
```

À présent, l'action peut utiliser cette nouvelle méthode pour retrouver les offres d'emploi actives.

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

Avantages du remaniement de code

Ce remaniement du code apporte plusieurs bénéfices indéniables par rapport au code précédent.

Tout d'abord, la logique de récupération des offres d'emploi actives est désormais à sa place dans le modèle. De ce fait, le contrôleur est considérablement allégé et rendu beaucoup plus lisible. D'autre part, ce remaniement a rendu la méthode `getActiveJobs()` réutilisable pour une

Les tests automatiques seront présentés au chapitre 8.

éventuelle prochaine utilisation. Enfin, le code est désormais disponible pour des tests unitaires.

Ordonner les offres suivant leur date d'expiration

Pour l'instant, toutes les offres sont récupérées et affichées dans l'*ordre de leur création*, c'est-à-dire en suivant la clé primaire. Or notre application doit montrer l'information la plus récente le plus souvent possible ; c'est pourquoi les offres doivent être ordonnées suivant leur date d'expiration – de la plus lointaine dans le futur à la plus proche de la date courante.

Le code suivant ordonne la liste des offres d'emploi suivant 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` détermine la clause `ORDER BY` de la requête SQL générée. Il est également possible d'utiliser `addOrderBy()` pour effectuer un tri sur plusieurs colonnes.

Classer les offres d'emploi selon leur catégorie

Parmi les objectifs définis au chapitre 2 figure celui-ci :

« Les offres d'emploi sont ordonnées par catégorie et ensuite par date de publication, des plus récentes aux plus anciennes ».

Jusqu'à présent, la catégorie d'une offre n'a pas été prise en compte. Or cette prise en compte fait partie du cahier des charges : la page d'accueil doit afficher les offres d'emploi *par catégorie*. Il est donc nécessaire de récupérer dans un premier temps toutes les catégories ayant au moins une offre d'emploi active. Pour ce faire, une méthode `getWithJobs()` doit être ajoutée à la classe `JobeetCategoryTable`.

Contenu du fichier 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();
  }
}
```

L'action `index` doit aussi être modifiée en conséquence :

Détail de la méthode executeIndex() du fichier apps/frontend/modules/job/actions/actions.class.php

```
public function executeIndex(sfWebRequest $request)
{
  $this->categories = Doctrine::getTable('JobeetCategory')
                                    ➥ ->getWithJobs();
}
```

Dans le template, les offres d'emploi actives sont désormais affichées en itérant à travers toutes les catégories.

Contenu du fichier apps/frontend/modules/job/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>
```

REMARQUE De l'utilité de la méthode magique __toString()

Pour afficher le nom de la catégorie dans le modèle, nous avons eu recours à

```
echo $category
```

Cela paraît-il étrange ? `$category` est un objet, comment `echo` peut-il afficher le nom de la catégorie ? La réponse a été donnée au chapitre 3 lorsque nous avons défini la méthode magique `__toString()` pour toutes les classes du modèle.

Pour que cela fonctionne, il est nécessaire d'ajouter la méthode `getActiveJobs()` à la classe `JobeetCategory`.

Détail de la méthode getActiveJobs() du fichier lib/model/doctrine/JobeetCategory.class.php

```
public function getActiveJobs()
{
  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.category_id = ?', $this->getId());

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

La méthode `JobeetCategory::getActiveJobs()` se sert de la méthode `Doctrine::getTable('JobeetJob')->getActiveJobs()` pour retrouver les offres d'emploi actives pour la catégorie donnée.

Le but, en faisant appel à `Doctrine::getTable('JobeetJob')->getActiveJobs()`, est de préciser la condition en fournissant une catégorie.

Un objet `Doctrine_Query` est passé à la place d'un objet `JobeetCategory`, c'est là en effet la meilleure manière d'encapsuler une condition générique.

La méthode `getActiveJobs()` a besoin de fusionner cet objet `Doctrine_Query` avec sa propre requête. C'est en fait très simple à réaliser puisque `Doctrine_Query` est lui-même un objet.

Détail de la méthode getActiveJobs() du fichier 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 le nombre de résultats affichés

Il y a un autre besoin fonctionnel à implémenter pour la page d'accueil de listage des offres d'emploi :

« Pour chaque catégorie, la liste montre *seulement les 10 premières offres d'emploi* et un lien permet de lister toutes les offres de cette dernière ».

C'est aussi simple que d'ajouter une méthode `getActiveJobs()` :

Méthode getActiveJobs() du fichier 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::getTable('JobeetJob')->getActiveJobs($q);
}
```

La clause `LIMIT` appropriée est pour l'instant codée en dur à l'intérieur du modèle. Or, il serait plus pertinent de rendre cette valeur configurable, comme nous l'avons vu plus haut dans ce chapitre. Pour ce faire, il suffit de changer le template afin de passer un nombre maximum d'offres d'emploi défini dans le fichier `app.yml`.

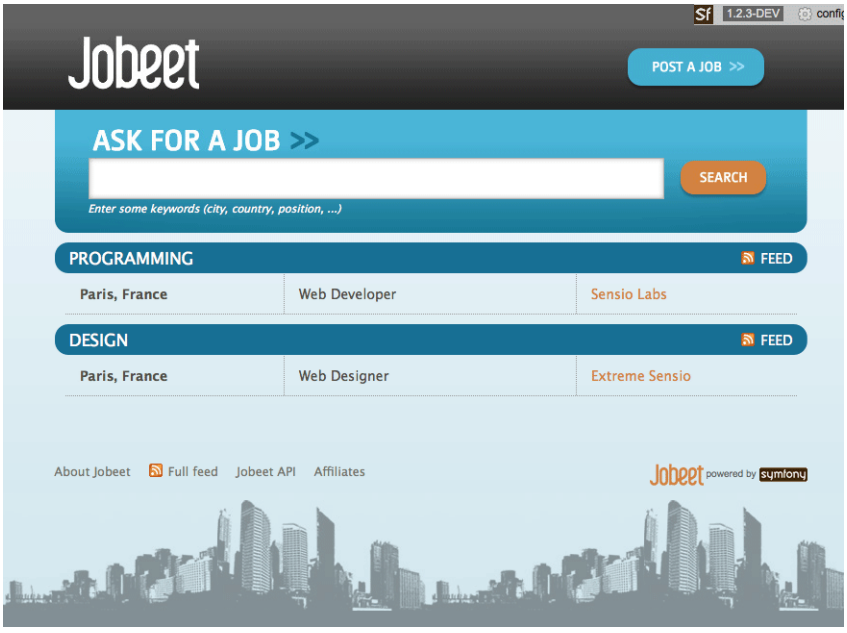


Figure 6–2
Classement des offres d'emploi par catégorie

Code à remplacer dans le fichier apps/frontend/modules/job/indexSuccess.php

```
<?php foreach ($category-
>getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')) as $i
=> $job): ?>
```

Puis, le nouveau paramètre doit être ajouté au fichier `app.yml` :

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

Modifier les données de test dynamiquement par l'ajout de code PHP

Mis à part réduire le paramètre `max_jobs_on_homepage` à 1, on ne distinguera aucune différence. Il est donc nécessaire d'ajouter un ensemble d'offres d'emploi aux données de test. Bien sûr, il est tout à fait possible de copier et coller une offre existante dix ou vingt fois à la main… mais il existe une bien meilleure façon de faire. La duplication est à éviter, même dans les fichiers de données.

Heureusement, les fichiers YAML de Symfony peuvent contenir du code PHP qui sera évalué juste avant l'analyse du fichier, comme le montre le fichier de données `jobs.yml` modifié :

```
JobeetJob:
# Starts at the beginning of the line (no whitespace before)
<?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 Indentation du code YAML

Attention à ne pas bousculer l'indentation de code YAML ! L'interpréteur syntaxique du YAML n'aimera pas le développeur si ce dernier désordonne l'indentation. Il faut toujours garder en tête les quelques astuces suivantes lorsque l'on ajoute du code PHP à un fichier YAML :

- les instructions `<?php ?>` doivent toujours être en début de ligne ou bien être encapsulées dans une valeur ;
- si une instruction `<?php ?>` termine une ligne, une nouvelle ligne (`\n`) doit explicitement être imprimée en sortie.

Lors du rechargement des données de test avec la tâche `doctrine:data-load`, on voit que seules dix offres d'emploi sont affichées sur la page d'accueil pour la catégorie `Programming`. Dans la capture d'écran suivante, le nombre maximum d'offres d'emploi a été fixé à cinq pour réaliser une image plus petite.

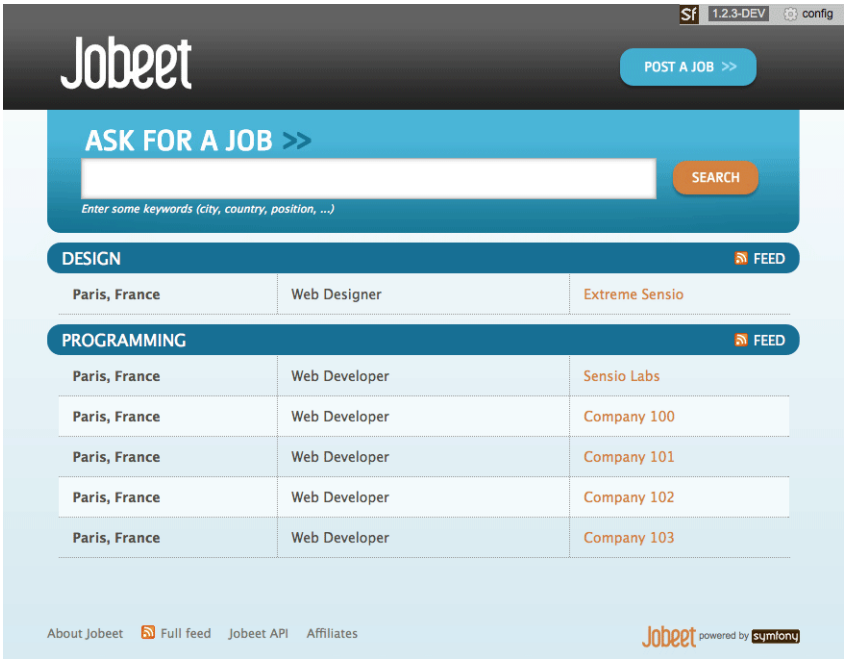


Figure 6–3
Limitation du nombre d'offres d'emploi par catégorie

Empêcher la consultation d'une offre expirée

Lorsqu'une offre d'emploi expire, même en ayant connaissance de l'URL, il ne devrait plus être possible d'y accéder. Il suffit d'essayer l'URL d'une offre expirée (en remplaçant l'`id` par l'`id` courant dans la base de données - `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 une offre d'emploi, l'application doit rediriger l'utilisateur vers une page d'erreur 404. Or comment réaliser cela, sachant qu'une offre est automatiquement retrouvée par sa route ?

Définition de la route job_show_user dans le fichier 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]
```

REMARQUE

À propos des versions de Symfony

Le paramètre `method_for_query` ne fonctionne pas pour les versions antérieures à Symfony 1.2.2.

Figure 6–4 Page d'erreur 404 d'une offre d'emploi expirée

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

Contenu du fichier 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();
  }

  // ...
}
```

Désormais, l'utilisateur qui tente d'accéder à une offre expirée est automatiquement redirigé vers une page d'erreur 404.

Créer une page dédiée à la catégorie

À présent, il serait pratique de disposer d'une page n'affichant que les offres d'une certaine catégorie (passée en paramètre), ainsi qu'un lien pour y accéder via la page d'accueil.

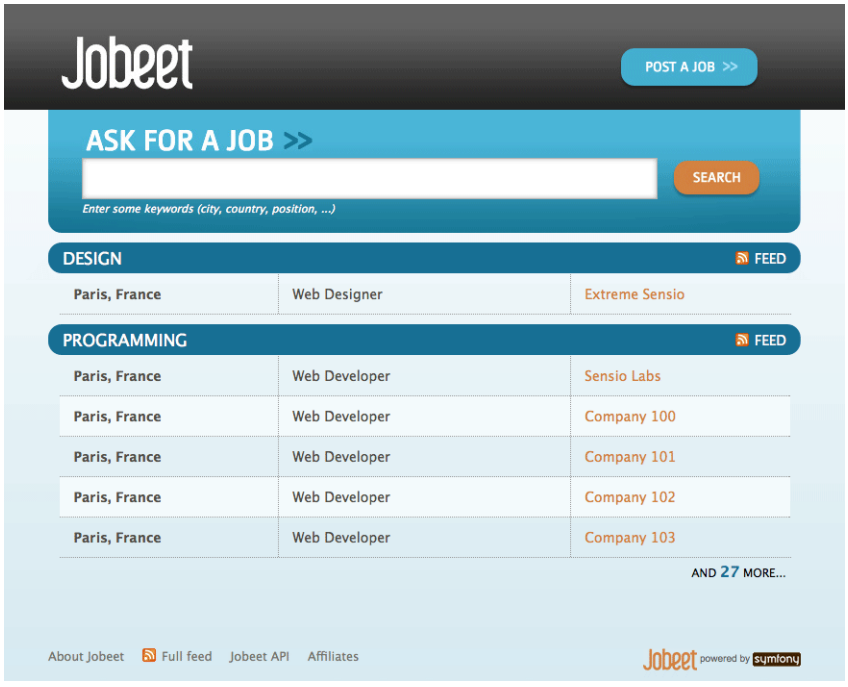
Mais attendez ! Arrêtons là ce sixième chapitre – même si nous n'avons pas tant travaillé – car vous avez suffisamment de connaissances pour implémenter vous-même cette fonctionnalité en guise d'exercice. La correction sera présentée au chapitre suivant…

En résumé…

Ce chapitre a abordé plus en détail ce qu'il est possible de réaliser à partir du modèle de données du framework Symfony. N'hésitez pas à travailler sur votre copie de travail locale de Jobeet ainsi qu'à utiliser la documentation de l'API en ligne, disponible sur le site officiel de Symfony à l'adresse http://www.symfony-project.org/api/1_2/. L'implémentation de la page de détail d'une catégorie sera, quant à elle, dévoilée tout au long du chapitre suivant.


symfony

chapitre 7


Jobeet
POST A JOB >>
ASK FOR A JOB >>
SEARCH
Enter some keywords (city, country, position, ...)
DESIGN
FEED
Paris, France
Web Designer
Extreme Sensio
PROGRAMMING
FEED
Paris, France
Web Developer
Sensio Labs
Paris, France
Web Developer
Company 100
Paris, France
Web Developer
Company 101
Paris, France
Web Developer
Company 102
Paris, France
Web Developer
Company 103
AND 27 MORE...
About Jobeet
Full feed
Jobeet API
Affiliates
Jobeet powered by symfony

Concevoir et paginer la liste d'offres d'une catégorie

MOTS-CLÉS :

- Routage
- Templates partiels
- Pagination

Les chapitres précédents ont montré toute l'importance du motif MVC et de l'intérêt de ne pas dupliquer le code dans un projet web. Le framework Symfony recèle encore bien d'autres outils pour satisfaire ce besoin dans la vue, notamment grâce aux templates partiels qui seront décrits dans ce septième chapitre.

Au cours du chapitre précédent, il a été montré que Symfony excelle dans différents domaines : les requêtes SQL avec Doctrine, les données de test, le routage, le débogage ainsi que la configuration sur mesure. Le dernier chapitre se clôturait sur un petit défi : mettre en place une « page dédiée à la catégorie ». Dans celui-ci, le but est de donner une solution à ce problème, en commençant par la présentation d'une potentielle implémentation.

Mise en place d'une route dédiée à la page de la catégorie

Déclarer la route category dans le fichier routing.yml

La première étape avant de démarrer l'implémentation de cette nouvelle page consiste à établir une nouvelle route propre à la catégorie. Dans la mesure où la catégorie est un objet Doctrine, le choix de la classe `sfDoctrineRoute` pour gérer la route dédiée en découle naturellement. Le code ci-dessous à ajouter au fichier de configuration `routing.yml` décrit la route propre à chaque catégorie.

Route à ajouter au début du fichier apps/frontend/config/routing.yml

```
category:
  url:      /category/:slug
  class:    sfDoctrineRoute
  param:    { module: category, action: show }
  options: { model: JobeetCategory, type: object }
```

Une route peut utiliser n'importe quelle colonne de son objet relatif en tant que paramètre. Elle peut également avoir recours à n'importe quelle autre valeur du moment qu'il existe un accesseur associé dans la classe de l'objet.

Implémenter l'accesseur getSlug() dans la classe JobeetJob

Du fait que le paramètre `slug` ne dispose d'aucune correspondance dans la table des catégories, la classe `JobeetCategory` doit se voir complétée d'un accesseur virtuel afin de rendre la route fonctionnelle.

Méthode getSlug() à ajouter au fichier lib/model/doctrine/JobeetCategory.class.php

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

MÉTHODE Implémentation d'une nouvelle fonctionnalité

Lorsque l'on démarre l'implémentation d'une nouvelle fonctionnalité, une bonne pratique consiste, dans un premier temps, à penser à l'URL, puis à créer la route associée. C'est d'ailleurs obligatoire quand les routes par défaut ont été supprimées.

Personnaliser les conditions d'affichage du lien de la page de catégorie

Intégrer un lien pour chaque catégorie ayant plus de dix offres valides

Pour l'instant, la page d'accueil de Jobeet ne dispose d'aucun lien pour se rendre directement sur la page de détail d'une catégorie. La page d'accueil est l'endroit opportun pour faire figurer un lien vers le détail de chaque catégorie listée. Néanmoins, pour ne pas surcharger inutilement cette page, seules les catégories ayant plus de dix offres d'emploi actives se verront ajouter un lien. Pour ce faire, le fichier `indexSuccess.php` du module `job` doit être modifié pour accueillir ce nouveau lien comme le montre le code ci-dessous.

```
<!-- 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>
```

Le lien sera ajouté à la page seulement s'il y a plus de 10 offres d'emploi à afficher pour la catégorie courante. Le lien contient le nombre d'offres non affichées. Afin que ce template puisse se comporter de la sorte, la méthode `countActiveJobs()` doit être implémentée dans la classe `JobeetCategory`.

Implémenter la méthode countActiveJobs() de la classe JobeetCategory

Le template `indexSuccess.php` du module `job` fait appel à la méthode `countActiveJobs()` de l'objet de la catégorie courante. Pour l'instant, cette méthode n'existe pas encore dans la classe `JobeetCategory`. Elle sert à compter le nombre d'offres d'emploi actives à l'instant t pour la catégorie. Le code ci-après donne le détail complet de cette nouvelle méthode.

Méthode countActiveJobs() à ajouter au fichier lib/model/doctrine/JobeetCategory.class.php

```
public function countActiveJobs()
{
  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.category_id = ?', $this->getId());

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

Implémenter la méthode countActiveJobs() de la classe JobeetCategoryTable

La méthode `countActiveJobs()` fait appel à une autre méthode `countActiveJobs()`, qui elle non plus n'existe pas dans la classe `JobeetJobTable`. Cette nouvelle méthode doit être capable de prendre un objet `Doctrine_Query` en argument afin de retourner le nombre d'offres d'emploi valides qui correspondent aux critères de cette requête. Il faut pour cela remplacer le code de `JobeetJobTable.class.php` par le code suivant :

Contenu du fichier 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 . '.expires_at DESC');

    return $q;
  }
}
```

Le code de la classe `JobeetJobTable` a été factorisé afin d'introduire une nouvelle méthode partagée `addActiveJobsQuery()` et de rendre ainsi le code plus DRY (Don't Repeat Yourself).

La méthode `countActiveJobs()` utilise directement la méthode `count()` plutôt que la méthode `execute()` suivie d'un comptage manuel des résultats, et ce pour des raisons évidentes de performance. En effet, recourir directement à la méthode `count()` évite la création et l'hydratation d'objets en mémoire ; seule la valeur résultante du dénombrement est renvoyée.

Plusieurs fichiers ont dû être modifiés dans le but d'implémenter cette nouvelle fonctionnalité. Cependant, le code écrit a systématiquement été placé dans la couche adéquate de l'application, notamment pour le rendre réutilisable ultérieurement. Tout au long du processus, des pièces de code existant ont été factorisées. Ce type de processus de travail est typique des projets et de la philosophie du framework Symfony.

BONNE PRATIQUE **La refactorisation de code**

La première fois qu'un bout de code est réutilisé, le copier peut paraître suffisant. Or, si on lui trouve un nouvel usage, cela implique qu'il faut refactoriser tous ces emplois dans une même fonction ou méthode partagée, comme il l'a été fait ici.

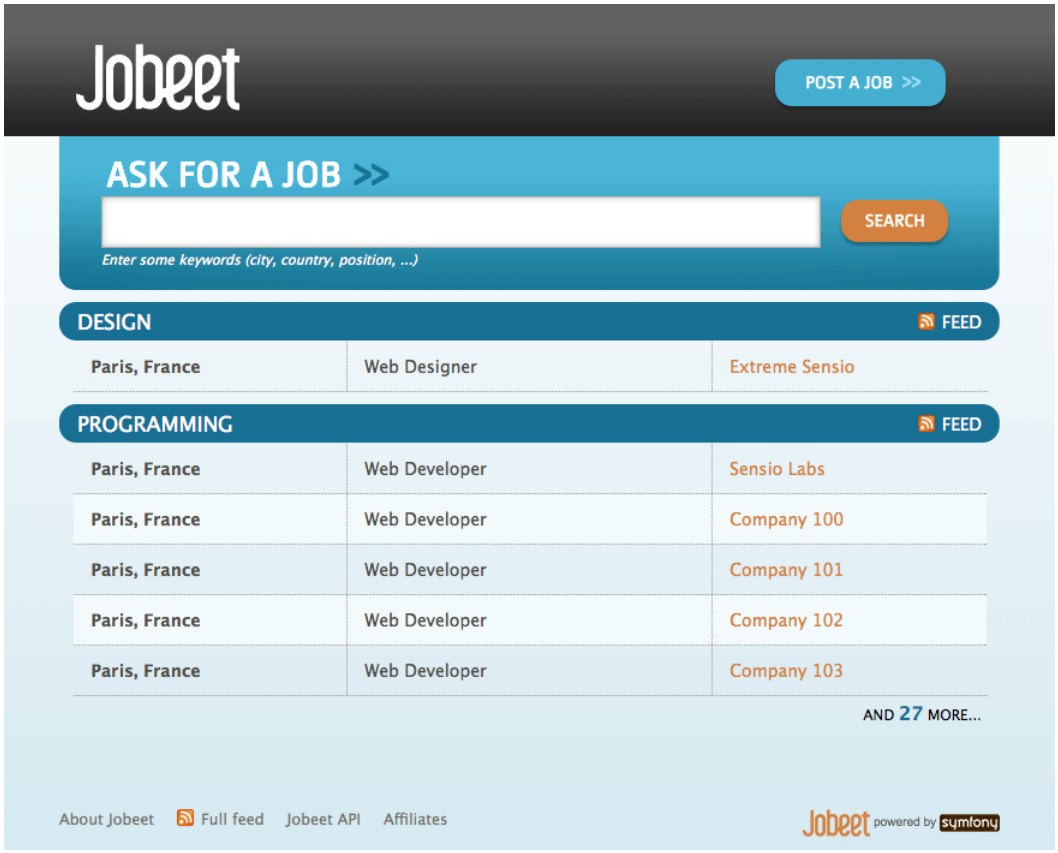


Figure 7–1
Page d'accueil de Jobeet

Mise en place du module dédié aux catégories

Générer automatiquement le squelette du module

L'application Jobeet doit se doter d'un nouveau module destiné à accueillir les spécificités des catégories. Il est bien sûr tentant de recourir à la tâche `doctrine:generate-crud` afin de construire un module complet de la même manière que pour le module job. Néanmoins, près de 90 % du code généré aurait été jeté au rebut. Pour cette raison, la création d'un module basique entièrement vide est réalisée au moyen de la tâche `generate:module`.

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

En accédant à la page d'une catégorie, la route de celle-ci doit trouver l'objet `JobCategory` associé à l'aide de la variable `slug` de la requête. Or, le slug n'est pas stocké dans la base de données, ce qui rend donc impossible la déduction d'une catégorie à partir du slug.

Ajouter un champ supplémentaire pour accueillir le slug de la catégorie

Afin de pouvoir identifier de manière unique une catégorie à partir d'un slug, la présence d'un champ `slug` dans la table `jobeet_category` est nécessaire. Grâce aux nombreux outils internes livrés avec Doctrine, la génération d'un slug à partir de la valeur d'un autre champ de la table SQL est entièrement automatisée. Pour ce faire, il suffit d'activer le comportement `Sluggable` pour le modèle `JobeetCategory` dans le schéma de description de la base de données.

Activation du comportement Sluggable dans le fichier config/doctrine/schema.yml

```
JobeetCategory:
  actAs:
    Timestampable: ~
    Sluggable:
      fields: [name]
  columns:
    name:
      type: string(255)
      notnull: true
```

À présent, la génération d'un slug unique à partir de la valeur du champ `name` est automatiquement gérée par Doctrine. De ce fait, la méthode `getSlug()` de la classe `JobeetCategory` n'a plus raison d'exister et conduit donc à son retrait. Pour finaliser la mise en place du slug, l'ensemble du modèle ainsi que la base de données doivent être reconstruits à l'aide de la tâche `doctrine:build-all-reload`. Au chargement des données initiales de test, le champ `slug` se verra automatiquement renseigné par Doctrine.

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

Création de la vue de détail de la catégorie

Mise en place de l'action executeShow()

La route `category` déclarée plus haut dans le fichier de configuration `routing.yml` s'appuie sur l'action `show` du module `category`. Pour l'instant, la méthode `executeShow()` ne figure pas dans la classe des actions du module. Le corps de cette méthode se résume à une seule et unique

CHOIX DE CONCEPTION Créer un nouveau module category dédié ou ajouter une action au module job ?

Pourquoi ne pas ajouter une action `category` au module `job` ? C'est en fait parce que le sujet principal est la catégorie elle-même. Pour cette raison, il semble plus naturel et logique de créer un module `category` dédié. Il y a en effet plus de sens à considérer la liste des catégories comme une entité à part entière, plutôt que comme une petite partie d'un module déjà existant. Savoir donner du sens à la conception générale de l'application et au code écrit n'est pas chose facile mais se révèle particulièrement important.

ligne de code qui a déjà été étudiée précédemment dans cet ouvrage. Il s'agit en effet de récupérer l'objet `JobeetCategory` à partir de sa route. Le code ci-dessous présente le contenu du fichier `actions.class.php` du module.

Contenu du fichier apps/frontend/modules/category/actions/actions.class.php

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

La méthode `index()` générée par défaut lors de la création du module a été volontairement supprimée du fichier car elle ne sera pas utilisée dans la suite du développement. De ce fait, son template associé `indexSuccess.php` peut à son tour être retiré du projet. Après la mise en œuvre de l'action `executeShow()`, il ne reste plus qu'à écrire le template correspondant `showSuccess.php`.

Intégration du template showSuccess.php associé

La finalisation de la nouvelle page de détail de la catégorie requiert évidemment l'écriture d'un fichier de template `showSuccess.php`. Ce dernier réutilise tel quel le tableau HTML du template `indexSuccess.php` du module `job` pour afficher la liste des offres d'emploi actives.

Contenu du fichier 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>
```

Bien qu'il soit entièrement fonctionnel, ce fichier dispose d'un inconvénient majeur : il duplique le code du template `indexSuccess.php` du module `job`. La philosophie du framework Symfony favorise autant que possible la séparation du code en plusieurs couches distinctes grâce au modèle MVC, mais aussi l'isolement du code dans le but d'éviter la duplication.

La duplication de code ne concerne pas seulement le code PHP, elle impacte également le code HTML, ce qui complexifie davantage les développements et la maintenance de l'application. La section suivante présente quelle solution technique le framework Symfony met en œuvre pour favoriser la factorisation et l'isolement du code HTML.

Isoler le HTML redondant dans les templates partiels

Découvrir le principe de templates partiels

Le template PHP `showSuccess.php` du module `category` copie l'intégralité de la balise `<table>` générant la liste d'offres d'emploi du fichier `indexSuccess.php`. Bien entendu, cette duplication du code va à l'encontre des principes fondamentaux et des bonnes pratiques défendus jusqu'à maintenant. Il est donc temps d'apprendre comment remédier à cette problématique.

Lorsqu'une petite partie d'un template a besoin d'être dupliquée pour être réutilisée ailleurs dans un autre, c'est le signe qu'il faut l'isoler dans un fichier à part. Dans Symfony, on parle de créer un template partiel (dit *partial*). Un partiel est un fragment de code d'un template qui a pour objectif d'être partagé par plusieurs autres templates. D'un point de vue technique, il s'agit en réalité de créer un fichier PHP dont le nom débute par un tiret souligné (ou underscore, caractère _) et de l'enregistrer dans les répertoires `templates/` du projet.

Création d'un template partiel _list.php pour les modules job et category

L'explication précédente a introduit la notion de templates partiels. Comment cela se présente-t-il plus concrètement dans un projet Symfony ? Comme il l'a été démontré juste avant, la génération de la liste des offres d'emploi d'une catégorie dans les templates `showSuccess.php` et `indexSuccess.php` est strictement identique. Le code dupliqué doit donc être isolé dans un template partiel `_list.php` du module `job` comme le montre le code ci-dessous.

Contenu du fichier 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>
```

Faire appel au partiel dans un template

L'utilisation d'un template partiel dans un template traditionnel est réalisée au moyen du helper Symfony `include_partial()`. Cette fonction accepte deux paramètres. Le premier est le nom du partiel à appeler. Il s'agit d'une chaîne de caractères qui se compose du nom du module dans lequel se trouve le partiel, puis d'une barre oblique / et enfin du nom du partiel auquel le tiret souligné de début est omis. Le second argument de `include_partial()` est un tableau associatif des variables à transmettre au partiel. Les clés du tableau correspondent au nom des variables du partiel, et les valeurs sont les variables à transmettre au partiel. Un bon exemple valant mieux qu'un long discours, le code ci-dessous résume le fonctionnement du helper `include_partial()`.

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

Les développeurs familiers du langage PHP se demandent certainement quelle est la véritable différence entre le helper `include_partial()` et un simple appel à `include()` ou `require()`. Le premier avantage concerne le

nommage des variables transmises au partiel. En effet, il n'est pas nécessaire que ces dernières aient le même nom que celles du partiel. C'est le tableau associatif passé en second argument qui se charge de faire la correspondance. D'autre part, le helper `include_partial()` a l'avantage de pouvoir mettre en cache le template partiel qu'il appelle.

Utiliser le partiel _list.php dans les templates indexSuccess.php et showSuccess.php

Le template partiel `_list.php` est prêt et n'attend qu'à être implémenté dans un template. De l'autre côté, les vues `showSuccess.php` et `indexSuccess.php` des modules `category` et `job` n'attendent qu'à être simplifiées grâce à ce dernier. Les bouts de code ci-dessous sont les appels au partiel à intégrer aux vues PHP précédentes à la place du code PHP qui génère la liste des offres d'emploi.

Code à ajouter au fichier apps/frontend/modules/job/templates/indexSuccess.php

```
<?php include_partial('job/list', array('jobs' => $category
->getActiveJobs(sfConfig::get('app_max_jobs_on_homepage')))) ?>
```

Code à ajouter au fichier apps/frontend/modules/category/templates/showSuccess.php

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

Paginer une liste d'objets Doctrine

Plus le site évolue, plus il grandit et plus son contenu s'enrichit. C'est d'autant plus vrai lorsque ce sont les utilisateurs eux-mêmes qui sont à la source de ce dernier. Pour cette raison, l'intégralité du contenu éditorial doit être clairement organisée dans le but de faciliter la navigation et l'expérience utilisateur. Cela débute naturellement par la pagination des listes de résultats récupérés de la base de données. Cette section explique pas à pas le processus de création d'une liste paginée d'offres d'emploi pour chaque catégorie.

Que sont les listes paginées et à quoi servent-elles ?

Paginer une liste d'objets en provenance d'une base de données est une tâche récurrente dans les projets informatiques. Les listes paginées servent par exemple à alléger le flux d'informations qui transitent sur le réseau, à optimiser le nombre de données affichées simultanément sur

l'écran de l'utilisateur, ou bien encore à réduire les temps de chargement des pages… En somme, elles améliorent le confort d'utilisation et l'expérience utilisateur.

Bien que ces listes soient courantes, elles n'en demeurent pas moins complexes et fastidieuses à mettre en œuvre car elles nécessitent le franchissement de plusieurs étapes successives, telles que la détermination de la page courante, le calcul du début de la position du curseur dans la liste de résultats, le comptage des objets à récupérer, la récupération des données, l'affichage des liens de pagination…

Préparer la pagination à l'aide de sfDoctrinePager

Heureusement, Symfony intègre un composant natif orienté objet qui facilite considérablement la création de listes paginées de résultats issus d'une base de données. Il s'agit de l'objet `sfDoctrinePager` qui a pour rôle de gérer dynamiquement la pagination d'une liste d'objets Doctrine. L'action `show` du module `category` est particulièrement concernée par la mise en œuvre d'un tel mécanisme dans la mesure où le nombre d'offres d'emploi publiées risque de croître à vive allure.

Initialiser la classe de modèle et le nombre maximum d'objets par page

Pour éviter de saturer inutilement la page et faire fuir l'utilisateur, la liste des offres doit être paginée, et des liens en bas de page doivent permettre de naviguer dans les offres de manière antéchronologique. Le code ci-dessous détaille la marche à suivre pour configurer la pagination d'une liste d'objets Doctrine à l'aide du composant `sfDoctrinePager`.

Définition d'un objet sfDoctrinePager dans 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();
}
```

Le constructeur de la classe `sfDoctrinePager` accepte deux arguments. Le premier est le nom de la classe modèle des objets à récupérer. Dans le cas présent, il s'agit de la classe `JobeetJob`. Le second est le nombre maximum d'objets présents sur chaque page.

Pour des raisons évidentes de personnalisation ultérieure, cette valeur est stockée dans une constante de configuration du fichier `app.yml`. Il sera ainsi plus aisé d'ajuster le nombre maximum d'objets par page si besoin.

Contenu du fichier apps/frontend/config/app.yml

```
all:
  active_days:          30
  max_jobs_on_homepage: 10
  max_jobs_on_category: 20
```

Spécifier l'objet Doctrine_Query de sélection des résultats

Par ailleurs, la méthode `setQuery()` de l'objet `sfDoctrinePager` reçoit en paramètre un objet de type `Doctrine_Query`. Celui-ci correspond effectivement à la requête SQL à exécuter pour rapatrier la liste d'objets Doctrine de la base de données.

Passer une `Doctrine_Query` en argument permet ainsi d'obtenir la liberté de créer des requêtes aussi bien simples que complexes. C'est là l'une des forces et toute la souplesse de l'ORM Doctrine. Dans le cas présent, l'objet `Doctrine_Query` est issu de la méthode `getActiveJobsQuery()` de la classe `JobeetCategory`. Cette méthode est implémentée quelques lignes plus bas.

Configurer le numéro de la page courante de résultats

Enfin, l'objet `sfDoctrinePager` a besoin de connaître le numéro de la page courante afin de déterminer quels enregistrements il doit récupérer et combien il y a de pages au total. La méthode `setPage()` accepte comme seul et unique argument le numéro de la page courante. Il est ici fourni à l'aide de la méthode `getParameter()` de l'objet `sfRequest`. Dans le cas présent, le numéro de la page en cours est transmis par l'URL dans la variable `page`. Le second argument de la méthode `getParameter()` correspond à la valeur par défaut à retourner si la variable `page` n'existe pas ou est nulle.

Initialiser le composant de pagination

Enfin, la méthode `init()` de l'objet `sfDoctrinePager` se charge d'initialiser la liste paginée. À ce stade, elle calcule le nombre de pages total, la position du curseur dans la liste ainsi que le nombre exact de résultats à paginer.

Simplifier les méthodes de sélection des résultats

Implémenter la méthode getActiveJobsQuery de l'objet JobeetCategory

Comme il l'a été décrit plus haut, le fonctionnement de l'objet `sfDoctrinePager` repose sur l'utilisation d'un objet de type `Doctrine_Query` pour effectuer le comptage du nombre total de résultats, ainsi que pour sélectionner les vingt objets par page. Pour l'instant, la méthode `getActiveJobsQuery()` de la classe `JobeetCategory` reste à implémenter. Cette dernière retourne la requête de sélection de toutes les offres actives de la catégorie courante triées par ordre antéchronologique suivant leur date d'expiration. Le code ci-dessous détaille l'implémentation de cette méthode.

Méthode getActiveJobsQuery() du fichier lib/model/doctrine/JobeetCategory.class.php

```
public function getActiveJobsQuery()
{
  $q = Doctrine_Query::create()
    ->from('JobeetJob j')
    ->where('j.category_id = ?', $this->getId());

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

Remanier les méthodes existantes de JobeetCategory

Maintenant que la méthode `getActiveJobsQuery()` est clairement définie, on peut se poser la question de savoir dans quelle mesure elle peut être réutilisée par d'autres méthodes de la classe. En effet, la requête SQL représentée par l'objet `Doctrine_Query` qu'elle renvoie est suffisamment générique et surtout commune aux méthodes `getActiveJobs()` et `countActiveJobs()` pour qu'il soit intéressant de procéder à un léger remaniement du code de ces dernières, afin de les simplifier en faisant en sorte qu'elles implémentent cette nouvelle méthode. À présent, les deux méthodes `getActiveJobs()` et `countActiveJobs()` se résument au code ci-après.

Refactorisation de méthodes dans le fichier lib/model/doctrine/JobeetCategory.class.php

```
public function getActiveJobs($max = 10)
{
  return $this->getActiveJobsQuery()->limit($max)->execute();
}

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

La dernière étape nécessaire à la finalisation de la pagination de la liste des offres d'emploi de la catégorie courante est bien évidemment l'intégration des liens de navigation à l'intérieur du template `showSuccess.php`. La partie suivante explique tout cela avant de clôturer ce chapitre.

Intégrer les éléments de pagination dans le template showSuccess.php

Le template `showSuccess.php` doit implémenter plusieurs éléments afin d'être complet. Les lignes qui suivent expliquent pas à pas les morceaux de code à ajouter ou modifier dans le template pour y parvenir. La première étape consiste à mettre à jour l'appel au partiel `_list.php`.

Passer la collection d'objets Doctrine au template partiel

Tout d'abord, le template `showSuccess.php` a besoin de mettre à jour l'appel au template partiel `_list.php` car la variable qui lui est transmise jusqu'à présent n'existe plus. En effet, la vue de l'action `show` reçoit maintenant de la part de l'action la variable `$pager` qui contient l'ensemble de la pagination, y compris la collection d'objets Doctrine à passer au template partiel.

Pour y parvenir, l'objet `sfDoctrinePager` intègre la méthode `getResults()` qui se charge d'exécuter la requête SQL de l'objet `Doctrine_Query`, puis d'hydrater la collection d'objets `JobeetJob` avant de la stocker dans une propriété privée et enfin de la retourner. Cette méthode renvoie un objet `Doctrine_Collection` qui est affecté à la clé `jobs` du tableau associatif comme le montre le code ci-dessous.

Appel au template partiel à remplacer dans le fichier apps/frontend/modules/category/templates/showSuccess.php

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

Afficher les liens de navigation entre les pages

L'étape suivante consiste à afficher les liens permettant à l'utilisateur de naviguer entre les différentes pages de la catégorie en cours. Une fois de plus, c'est grâce au composant `sfDoctrinePager` et à ses méthodes très pratiques que l'on est capable de générer un système de navigation complet sans se poser de questions.

Le code suivant est à placer tout en bas de la vue `showSuccess.php`. Il génère un système de pagination intégrant les liens pour accéder aux première et dernière pages, aux pages précédente et suivante, ainsi qu'aux pages intermédiaires en prenant garde à désactiver le lien de la page sur laquelle se trouve l'utilisateur. La description de chacune des méthodes de l'objet de pagination appelé, est expliquée plus loin.

Création d'un système de pagination complet dans le fichier apps/frontend/modules/category/templates/showSuccess.php

```
<?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" />
    </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; ?>
```

Afficher le nombre total d'offres publiées et de pages

Enfin, l'idéal est d'informer l'utilisateur du nombre total d'offres d'emploi publiées dans la catégorie courante ainsi que le nombre maximum de pages qu'il est en mesure de parcourir. Un petit indicateur supplémentaire lui permet de savoir à tout moment sur quelle page il se trouve par rapport aux autres comme le montre le bout de code ci-dessous à placer en bas du template.

Indicateurs du nombre total de résultats et de pages dans le fichier apps/frontend/modules/category/templates/showSuccess.php

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

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

Description des méthodes de l'objet sfDoctrinePager utilisées dans le template

L'objet `sfDoctrinePager` fournit un certain nombre de méthodes particulièrement utiles pour retrouver des informations sur l'état de la pagination. Cela va du numéro de la page courante au nombre total de résultats, en passant par les numéros des pages suivantes et précédentes. Toutes les méthodes listées et décrites ci-dessous figurent dans le template `showSuccess.php` et ont permis l'élaboration de la pagination de ce dernier.

- `getResults()` : retourne une collection d'objets Doctrine pour la page courante ;
- `getNbResults()` : retourne le nombre total de résultats ;
- `haveToPaginate()` : retourne vrai s'il y a plus qu'une page ;
- `getLinks()` : retourne la liste des liens des pages à afficher ;
- `getPage()` : retourne le numéro de la page courante ;
- `getPreviousPage()` : retourne le numéro de la page précédente ;
- `getNextPage()` : retourne le numéro de la page suivante ;
- `getLastPage()` : retourne le numéro de la dernière page.

Code final du template showSuccess.php

Contenu du fichier 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" />
    </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 $pager->getNbResults() ?></strong> jobs in
this category
```

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

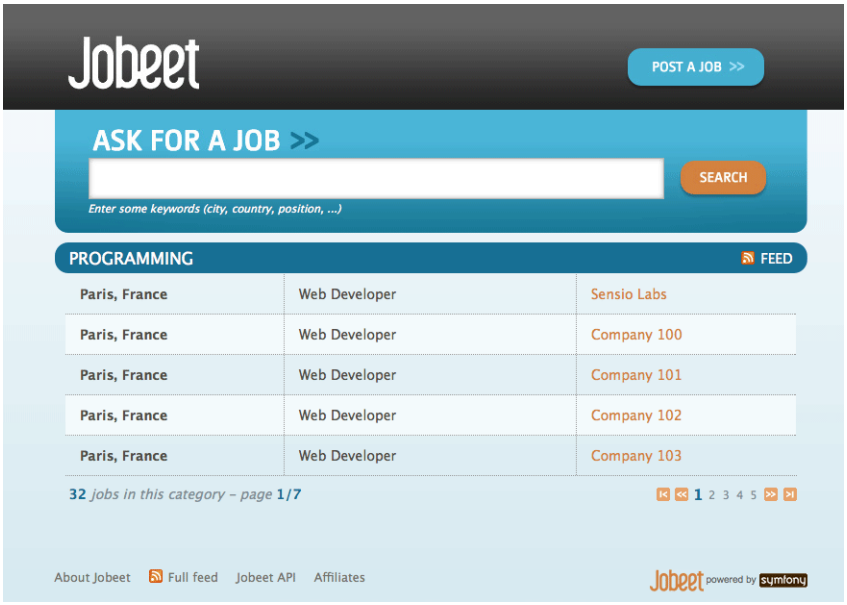


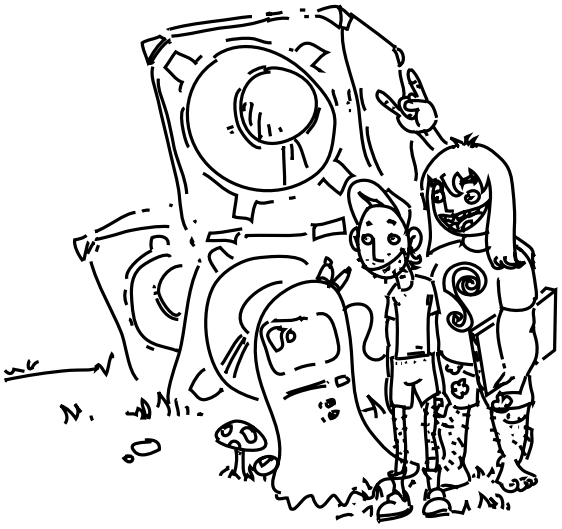
Figure 7–2
Résultat de la pagination de la liste des offres d'emploi

En résumé…

Si vous aviez travaillé sur votre propre implémentation et si vous avez ressenti que vous n'avez pas appris grand chose après ces quelques pages, cela signifie que vous vous êtes progressivement habitué à la philosophie du framework Symfony. Le processus d'ajout d'une nouvelle fonctionnalité à un site web Symfony reste toujours le même : songer d'abord aux URLs, créer les actions, mettre à jour le modèle et enfin écrire quelques templates. Et, si vous pouvez appliquer quelques bonnes pratiques de développement au passage, vous maîtriserez très vite le framework. Le chapitre suivant s'intéresse à un sujet clé du développement d'applications web professionnelles, à savoir la notion de tests unitaires et fonctionnels automatisés. Le prochain chapitre se focalise sur les tests unitaires tandis que les tests fonctionnels seront présentés en détail dans le chapitre 9…

chapitre 8

```
~/work/jobeet $ php symfony test:unit Jobeet
1..1
ok 1 - This test always passes.
 Looks like everything went fine.
~/work/jobeet $
```



Les tests unitaires

MOTS-CLÉS :

- Tests unitaires
- Framework Lime
- Couverture de code

Trop souvent négligés, oubliés, voire laissés de côté faute de temps, les tests unitaires sont pourtant des garants de la qualité et de la pérennité d'une application web.

Symfony intègre nativement un framework de tests unitaires et fonctionnels. Ce huitième chapitre se consacre à la découverte du premier et à l'écriture des premiers tests automatisés de Jobeet.

Les deux précédents chapitres ont permis de revoir l'ensemble des fonctionnalités acquises jusqu'à présent, d'en personnaliser certaines mais aussi d'en ajouter d'autres.

Ce chapitre aborde quelque chose de complètement différent : les **tests** automatisés. Dans la mesure où ce sujet est particulièrement large, deux chapitres entiers y sont consacrés afin de pouvoir couvrir l'essentiel des fondamentaux.

Présentation des types de tests dans Symfony

Il existe deux types de tests automatisés dans Symfony : les tests unitaires et les tests fonctionnels.

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

En revanche, les tests fonctionnels s'assurent que l'application résultante se comporte correctement dans sa globalité.

Dans un projet Symfony, tous les tests se situent dans le répertoire `test/` du projet. Il contient deux sous-dossiers : un pour les tests unitaires (`test/unit/`) et l'autre pour les tests fonctionnels (`test/functional/`).

Ce chapitre n'aborde que les tests unitaires tandis que les tests fonctionnels seront présentés au chapitre suivant.

De la nécessité de passer par des tests unitaires

Écrire des tests unitaires est probablement l'une des bonnes pratiques les plus importantes du développement web à mettre en application. En effet, les développeurs ne sont pas toujours sensibilisés à tester leur travail. Ainsi, cela permet de soulever plusieurs interrogations : faut-il écrire des tests avant d'implémenter une fonctionnalité ? Quels outils sont nécessaires pour tester efficacement ? Les tests ont-ils besoin de couvrir tous les cas possibles ? Comment s'assurer que tout l'ensemble du projet est bien testé ? Cependant, la toute première question qui se pose est généralement beaucoup plus triviale : par où commencer ?

Si tester massivement est toujours une bonne pratique, l'approche de Symfony n'en demeure pas moins pragmatique : il est en effet préférable d'avoir seulement quelques tests sous la main plutôt que rien du tout… Le projet a-t-il déjà beaucoup de code non testé ? Il n'est pas nécessaire d'avoir une suite complète de tests pour profiter des avantages des tests automatisés. Dans un premier temps, il est bon de commencer par ajouter des tests à chaque fois qu'un bogue est découvert. Au fil du temps, le code s'améliore, la couverture de code s'élargit, et la confiance en celui-ci croît. Tout cela est rendu possible grâce à cette approche pragmatique. L'étape suivante consiste donc à écrire des tests lors de l'implémentation de nouvelles fonctionnalités. Ces derniers se montreront très vite indispensables dans la suite du projet !

Le problème avec la plupart des librairies de tests reste leur courbe d'apprentissage particulièrement raide. C'est pourquoi Symfony fournit une librairie de tests très simple, **lime**, afin de simplifier l'écriture de tests.

Si ce chapitre décrit en profondeur la librairie intégrée lime, rien n'empêche d'en utiliser une autre comme l'excellent PHPUnit.

Présentation du framework de test lime

Initialisation d'un fichier de tests unitaires

Tous les tests écrits à partir du framework lime débutent avec le même code :

```
require_once dirname(__FILE__).'/../bootstrap/unit.php';
$t = new lime_test(1, new lime_output_color());
```

Tout d'abord, le fichier d'initialisation `unit.php` est inclus afin de charger la configuration du projet et de la bibliothèque lime. Puis, un nouvel objet `lime_test` est créé, et le nombre de tests à exécuter est passé comme argument au constructeur de la classe. Le nombre de tests unitaires prévus permet à lime d'imprimer un message d'erreur en sortie au cas où trop peu de tests seraient exécutés (par exemple, lorsqu'un test provoque une erreur fatale PHP).

Découverte des outils de tests de lime

« Tester les fonctionnements d'une méthode ou bien d'une fonction, c'est appeler cette dernière avec des points d'entrée prédéfinis, puis comparer la valeur qu'elle retourne avec la valeur de la sortie attendue ». Cette comparaison détermine alors si un test passe ou bien s'il échoue.

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

Tableau 8–1 Liste des méthodes disponibles de l'objet lime_test

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

Le fait que lime définisse autant de méthodes de test peut paraître étrange dans la mesure où tous les tests peuvent être écrits à l'aide de la méthode `ok()`. L'avantage de ces méthodes alternatives réside dans les messages d'erreur beaucoup plus explicites qu'elles produisent lorsque le test échoue. De plus, elles permettent de faciliter la relecture des tests unitaires. Le tableau suivant liste quelques méthodes utiles de l'objet `lime_test`.

Tableau 8–2 Liste des méthodes de test de l'objet lime_test

Nom de la méthode	Description
`fail()`	Échoue toujours – pratique pour tester des exceptions
`pass()`	Passe toujours – pratique pour tester des exceptions
`skip($msg, $nb_tests)`	Compte comme `$nb_tests` – pratique pour les tests conditionnels
`todo()`	Compte comme un test – pratique pour les tests non encore écrits

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

Exécuter une suite de tests unitaires

Tous les tests unitaires sont stockés dans le répertoire `test/unit`. Par convention, les tests sont nommés avec le nom de la classe (ou de la fonction) qu'ils testent, suffixés par `Test`. S'il est toujours possible d'organiser les fichiers du répertoire `test/unit` à sa guise, il est conseillé de répliquer la structure du répertoire `lib/`.

Afin d'illustrer la mise en application des tests unitaires, la classe `Jobeet` sera testée via un nouveau fichier `test/unit/JobeetTest.php` qui contient le code suivant :

Contenu du fichier test/unit/JobeetTest.php

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

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

Il existe deux manières de lancer les tests. La première consiste à exécuter directement le fichier PHP à l'aide du binaire `php` :

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

La seconde méthode permet quant à elle d'exécuter la suite de tests unitaires grâce à la commande `test:unit`.

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

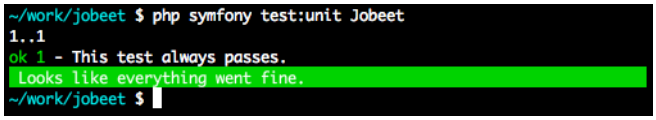


Figure 8–1
Résultat d'exécution d'une suite de tests unitaires

Sous Windows, la ligne de commande ne permet malheureusement pas de mettre en évidence les résultats des tests en vert et rouge.

Tester unitairement la méthode slugify()

Afin d'illustrer plus concrètement les principes de tests unitaires, c'est la méthode `Jobeet::slugify()` qui sera testée dans un premier temps. Il s'agit en effet de montrer les outils du framework lime ainsi que les bonnes pratiques à mettre en œuvre lorsque l'on teste du code.

Déterminer les tests à écrire

La méthode `slugify()` a été créée au cinquième jour pour nettoyer une chaîne afin que celle-ci soit utilisée en toute sécurité dans une URL. La modification de la chaîne d'origine consiste en quelques transformations basiques comme la conversion des caractères non ASCII par un tiret (-) ou le passage de la chaîne en lettres minuscules.

Tableau 8–3 Tableau de présentation du fonctionnement de la méthode slugify

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

Écrire les premiers tests unitaires de la méthode

Il est temps de contrôler que la méthode `slugify()` actuelle de la classe `Jobeet` réalise correctement la transformation de la chaîne de caractères qui lui est passée en argument. Pour ce faire, le contenu du fichier de test doit être remplacé par celui-ci :

Contenu du fichier test/unit/JobeetTest.php

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

$t = new lime_test(6, new lime_output_color());

$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');
```

En prêtant attention aux tests écrits, il est important de constater que chaque ligne teste seulement un cas particulier. C'est un principe dont il faut toujours se souvenir lors de l'écriture de tests unitaires. Il est important de tester une seule chose à la fois !

Il ne reste maintenant plus qu'à exécuter le fichier de tests. Si tous les tests passent comme prévu, la « barre verte » fera son apparition. Dans le cas contraire, ce sera la « barre rouge » qui alertera que certains tests ont échoué.

Figure 8–2
Résultat d'exécution des tests unitaires de Jobeet::slugify()

Si un test échoue, la sortie donnera quelques informations pour en connaître les raisons. Cependant, si le fichier contient une centaine de tests, il sera probablement difficile d'identifier rapidement le comportement inhabituel.

Commenter explicitement les tests unitaires

Toutes les méthodes de tests de lime prennent une chaîne comme dernier argument qui sert de description pour un test. C'est là un outil très pratique dans la mesure où cela oblige le développeur à décrire lui-même la portion de code testée. Cela peut également servir de documentation du comportement attendu d'une méthode. Le code ci-dessous illustre un

exemple d'une série de tests de la méthode `slugify()`, commentée par de courtes phrases :

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

$t = new lime_test(6, new lime_output_color());

$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 $
```

Figure 8–3
Commentaires descriptifs des tests unitaires de Jobeet::slugify()

REMARQUE **La couverture de code**

Lorsque l'on écrit des tests, il arrive souvent d'en oublier pour certaines parties du code. Pour aider à vérifier que tout le code est bien testé, Symfony fournit la tâche `test:coverage`. En passant comme arguments de cette commande un fichier de test, ou un répertoire et un fichier, ou encore un répertoire de bibliothèque, cette dernière renverra en résultat le taux de couverture global du code.

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

Si l'on souhaite savoir quelles lignes ne sont pas couvertes par les tests, il suffit de passer l'option `--detailed`.

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

Toutefois, il est très important de garder à l'esprit que lorsque la tâche indique que le code est complètement testé unitairement, cela signifie seulement que chaque ligne a été exécutée mais pas que tous les cas possibles ont été testés.

Comme la tâche `test:coverage` dépend de `XDebug` pour récolter ces informations, l'extension doit impérativement être installée et activée en premier lieu.

La chaîne de description du test est aussi un outil précieux lorsque l'on essaie de savoir ce qu'il faut tester. En effet, elle doit permettre d'expliquer succinctement comment doit se comporter la fonction ou la méthode testée. Ce bref résumé du test suit un formalisme bien défini pour faciliter la lecture des résultats. Les tests de méthodes statiques débutent tous par `::` suivis du nom de la méthode, tandis que les méthodes classiques d'instance sont préfixées par `->`.

Implémenter de nouveaux tests unitaires au fil du développement

Ajouter des tests pour les nouvelles fonctionnalités

Le slug d'une chaîne vide est une chaîne vide. Un test unitaire peut le vérifier. Or, une chaîne vide dans une URL n'est pas conseillée. La méthode `slugify()` doit être modifiée afin qu'elle retourne la chaîne `n-a` en cas de chaîne vide.

La résolution de ce cas critique peut être réalisée en écrivant le test avant l'implémentation de la fonctionnalité ou bien après. C'est simplement une question de goût. Néanmoins, écrire le test avant permet d'avoir la certitude que le code implémente ce qui a été prévu.

```
$t->is(Jobeet::slugify(''), 'n-a', '::slugify() converts the
empty string to n-a');
```

À présent, en exécutant la suite de tests unitaires, la barre rouge fait son apparition. Si ce n'est pas le cas, soit la fonctionnalité est déjà implémentée, soit le test ne contrôle pas ce qu'il est supposé tester. Le code suivant corrige le slug résultant lorsqu'une chaîne de caractères vide est passée en argument.

Début de la méthode slugify() de la classe lib/Jobeet.class.php

```
static public function slugify($text)
{
  if (empty($text))
  {
    return 'n-a';
  }

  // ...
}
```

Le test doit maintenant passer comme prévu, et afficher la barre verte tant attendue, à condition d'avoir pensé à mettre à jour le nombre de tests planifiés dans le constructeur de l'objet `lime_test`. Si ce n'est pas le cas, il y aura un message informant que 6 tests ont été planifiés mais que 7 ont été exécutés au total. Garder le compteur de tests planifiés à jour est important car il permet de tenir le développeur informé si le script de test « meurt » trop tôt.

Ajouter des tests suite à la découverte d'un bug

Il est possible qu'un utilisateur du site rapporte un bug lors de sa navigation sur le site : les liens présentant certaines offres d'emploi pointent vers une page d'erreur 404. Cette erreur provient du fait que des offres possèdent un slug dont le nom de la société, le poste ou la situation géographique est vide. Après vérification, aucun champ de la base de données n'a pourtant été laissé vide.

Ce bogue survient en fait à cause d'un problème dans la méthode statique `slugify()`. En effet, lorsque la chaîne passée en paramètre ne contient que des caractères dits « non-ASCII », cette méthode convertit la chaine en une chaîne vide. Un réflexe bien naturel serait de corriger directement la méthode `slugify()` et ainsi de ne plus en entendre parler. Ce réflexe est à oublier dans un projet implémentant un framework de tests unitaires. À défaut de corriger directement le bogue identifié, il va d'abord s'agir de créer un test, afin de s'assurer par la suite que ce problème ne se produise plus.

```
$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 $
```

Figure 8–4
Résultats des tests unitaires de Jobeet::slugify() suite au bug découvert

Ce n'est qu'après avoir vérifié que le test ne passe pas qu'il sera nécessaire d'éditer la classe `Jobeet` et de déplacer le contrôle de la chaîne vide à la fin de la méthode comme ci-dessous.

```
static public function slugify($text)
{
  // ...

  if (empty($text))
  {
    return 'n-a';
  }

  return $text;
}
```

Désormais, le nouveau test ainsi que tous les autres passent correctement. La méthode `slugify()` avait bel et bien un bogue malgré une couverture de code de 100 %.

Il est impossible de penser à tous les cas de bogues possibles lors de l'écriture de tests. Néanmoins, lorsqu'un problème est découvert, un test pour celui-ci doit, dans l'idéal, être écrit avant de corriger le code. Cela signifie aussi que le code gagnera de plus en plus en qualité, ce qui est toujours une bonne chose. De plus, il ne faut pas oublier que le temps gagné sur le développement de grosses applications est considérable. En n'ayant plus à tester ce genre de cas de figure à la main, le développeur peut se concentrer davantage sur des portions de code plus critiques.

Implémenter une meilleure méthode slugify

Il est important de rappeler que Symfony a été créé par des Français. De ce fait, les chaînes contenant des caractères accentués sont monnaie courante et doivent donc être pris en charge par la méthode `slugify()`. Pour ce faire, la première étape consiste à écrire un nouveau test unitaire avec une chaîne contenant des caractères accentués.

```
$t->is(Jobeet::slugify('Développeur Web'), 'developpeur-web',
'::slugify() removes accents');
```

Bien évidemment, ce test doit échouer car les caractères accentués sont automatiquement remplacés par des tirets et non par leur correspondant non accentué respectif. C'est un problème récurent appelé « translittération ». Heureusement, si la librairie « `iconv` » est installée sur le serveur, elle réalisera automatiquement le travail permettant d'éviter ce bogue.

```
// code dérivé de 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;
}
```

Pour éviter ce genre de problèmes, il est important de sauvegarder les fichiers PHP avec l'encodage UTF-8 dans la mesure où c'est l'encodage par défaut de Symfony et qu'il est le seul utilisé par « iconv » pour faire de la translittération.

Sachant que ce problème se posera uniquement si la librairie iconv n'est pas disponible, le test ne sera exécuté qu'à cette condition. Aussi, le fichier de test doit être modifié.

```
if (function_exists('iconv'))
{
  $t->is(Jobeet::slugify('Développeur Web'), 'developpeur-web',
                          '::slugify() removes accents');
}
else
{
  $t->skip('::slugify() removes accents - iconv not installed');
}
```

Implémentation des tests unitaires dans le framework ORM Doctrine

Configuration de la base de données

Tester unitairement un modèle Doctrine est un peu plus complexe dans la mesure où une connexion à la base de données est nécessaire.

Bien sûr, il conviendrait d'utiliser la connexion à la base de données utilisée en environnement de développement, mais c'est une bonne habitude à prendre que de créer une base de données dédiée aux tests. De cette manière, il est possible d'utiliser des données source de bogues dans les précédents développements et ainsi de vérifier la validité des tests.

Au cours du premier chapitre, les environnements ont été introduits comme un moyen de varier les paramètres d'une application. Par défaut, tous les tests sont exécutés dans l'environnement de test ; une base de données doit donc être configurée pour ce dernier.

```
$ php symfony configure:database --name=doctrine
  ➥ --class=sfDoctrineDatabase --env=test
  ➥ "mysql:host=localhost;dbname=jobeet_test" root mYsEcret
```

EN COULISSE Organisation du fichier de configuration de Symfony

Le fichier de configuration `config/databases.yml` est très instructif. Il permet de constater comment Symfony simplifie le changement de configuration en fonction d'un environnement.

L'option `env` indique à la tâche que la configuration de la base de données est valable uniquement pour l'environnement de test. Lorsque cette commande a été utilisée au chapitre 3, l'option `env` n'avait pas été utilisée, c'est pourquoi la configuration a été appliquée pour tous les environnements.

Maintenant que la base de données a été configurée, il est nécessaire de l'initialiser en utilisant la commande `doctrine:insert-sql`.

```
$ mysqladmin -uroot -pmYsEcret create jobeet_test
$ php symfony doctrine:insert-sql --env=test
```

IMPORTANT Principes de configuration dans Symfony

Le quatrième chapitre montrait que des paramètres provenant de fichiers de configuration pouvaient être définis à plusieurs niveaux.
Ces paramètres de configuration peuvent également être dépendants d'un environnement. C'est vrai pour la plupart des fichiers de configuration utilisés jusqu'à présent : `databases.yml`, `app.yml`, `view.yml` et `settings.yml`. La clé principale présente dans ces fichiers correspond à l'environnement et indique que les paramètres sont définis pour chaque environnement.

```
# 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
```

Mise en place d'un jeu de données de test

Maintenant que la base de données dédiée aux tests a été mise en place, il est temps de trouver une manière pour y charger des informations de test. Lors du troisième jour, la commande `doctrine:data-load` a été abordée, mais pour les tests, les données devront être rechargées à chaque exécution afin de fixer la base de données dans un état connu.

La tâche `doctrine:data-load` utilise intérieurement la méthode `Doctrine::loadData` pour charger les données.

```
Doctrine::loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

La classe globale `sfConfig` peut être utilisée pour obtenir le chemin complet vers un sous-répertoire du projet. L'utiliser permet notamment de pouvoir personnaliser la structure des dossiers par défaut.

La méthode `loadData()` prend un répertoire ou bien un nom de fichier comme premier argument. Elle peut aussi recevoir un tableau de dossiers et/ou de fichiers.

Quelques données initiales ont déjà été créées dans le répertoire `data/fixtures/`. Pour les tests, les données utilisées seront déposées dans le répertoire `test/fixtures/`. Ces fichiers de données seront ensuite utilisés par Doctrine pour les tests unitaires et fonctionnels.

Pour l'instant, il suffit simplement de copier les fichiers du répertoire `data/fixtures/` dans le répertoire `test/fixtures`.

Vérifier l'intégrité du modèle par des tests unitaires

Initialiser les scripts de tests unitaires de modèles Doctrine

Voici quelques tests unitaires pour la classe de modèle `JobeetJob`. Comme tous les tests unitaires Doctrine débuteront par le même code, et pour respecter le principe DRY, un fichier `Doctrine.php` dans le répertoire de test `bootstrap/` doit être créé. Ce dernier contient le code suivant :

Contenu du fichier test/bootstrap/Doctrine.php

```
include(dirname(__FILE__).'/unit.php');

$configuration = ProjectConfiguration::getApplicationConfiguration( 'frontend', 'test', true);

new sfDatabaseManager($configuration);

Doctrine::loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

Le script est suffisamment explicite de lui-même :

- comme pour les *front controllers*, un objet de configuration est initialisé pour l'environnement de test :

```
$configuration =
ProjectConfiguration::getApplicationConfiguration( 'frontend',
'test', true);
```

- puis un gestionnaire de base de données est créé. Ce dernier initialise la connexion Doctrine en chargeant le fichier de configuration `databases.yml` :

```
new sfDatabaseManager($configuration);
```

- enfin, les données de tests sont chargées en base de données grâce à `Doctrine::dataLoad()` :

```
Doctrine::loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

EN COULISSE
Connexion au serveur par Doctrine

Doctrine se connecte à la base de données uniquement s'il y a des requêtes SQL à exécuter.

Tester la méthode getCompanySlug() de l'objet JobeetJob

Maintenant que tout est en place, les tests de la classe `JobeetJob` peuvent démarrer. Pour commencer, un fichier `JobeetJobTest.php` doit être créé dans le répertoire `test/unit/model`.

Contenu du fichier test/unit/model/JobeetJobTest.php

```
include(dirname(__FILE__).'/../../bootstrap/Doctrine.php');

$t = new lime_test(1, new lime_output_color());

$t->comment('->getCompanySlug()');
$job = Doctrine::getTable('JobeetJob')->createQuery()-
>fetchOne();
$t->is($job->getCompanySlug(), Jobeet::slugify($job-
>getCompany()), '->getCompanySlug() return the slug for the
company');
```

Au passage, il est important de remarquer que le test porte seulement sur la méthode `getCompanySlug()`, aucune vérification n'étant effectuée pour savoir si le slug est correct ou non. En effet, c'est un test qui a déjà été effectué ailleurs.

MÉTHODE **Tenir à jour le compteur de tests planifiés**

Chaque fois que des tests sont ajoutés, il ne faut pas oublier de mettre à jour le compteur de tests planifiés dans le constructeur de la classe `lime_test`. Pour `JobeetJobTest`, il suffit de remplacer 1 par 3.

Tester la méthode save() de l'objet JobeetJob

Écrire des tests pour la méthode `save()` est légèrement plus complexe. En effet, il s'agit ici de vérifier que la création de l'objet s'est bien produite mais surtout que les données insérées dans la base de données sont bien celles auxquelles on s'attend.

```
$t->comment('->save()');
$job = create_job();
$job->save();
$expiresAt = date('Y-m-d', time() + 86400 * sfConfig::get('app_active_days'));
$t->is(date('Y-m-d', strtotime($job->getExpiresAt())), $expiresAt, '->save() updates expires_at
if not set');

$job = create_job(array('expires_at' => '2008-08-08'));
$job->save();
$t->is(date('Y-m-d', strtotime($job->getExpiresAt())), '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::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;
}
```

Implémentation des tests unitaires dans d'autres classes Doctrine

Il est dès maintenant possible d'ajouter des tests unitaires pour chacune des classes Doctrine. Le processus d'écriture de tests unitaires étant facile à assimiler, la tâche devrait donc être tout aussi aisée.

Lancer l'ensemble des tests unitaires du projet

La tâche `test:unit` peut aussi servir à lancer tous les tests du projet :

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

Elle indique en sortie si chaque fichier de tests est passé ou bien a échoué.

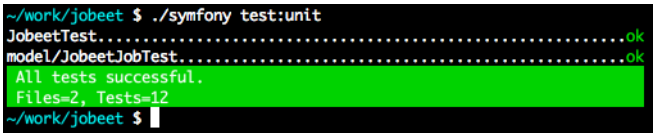


Figure 8–5
Résultats d'exécution de tous les tests unitaires du projet

Si la tâche `test:unit` retourne le statut `dubious` pour un fichier de tests, cela indique que le script s'est interrompu avant la fin. Exécuter le fichier de tests seul donnera le message d'erreur exact.

En résumé…

Tester une application est une chose nécessaire ; et pourtant, certains lecteurs auront sans doute été tentés de faire l'impasse sur ce chapitre… Nous sommes ravis de constater que ce n'est pas le cas !

Bien sûr, seule une pratique intense de Symfony vous en révèlera les fonctionnalités les plus intéressantes, ainsi que la philosophie de développement et les bonnes pratiques qu'il induit. Tester fait justement partie de ces pratiques vertueuses et un jour ou l'autre, les tests unitaires sauveront vos applications du désastre.

Les tests, en effet, garantissent la solidité du code et offrent la liberté de le remanier sans risque de tout casser. Ils sont de véritables garde-fous car ils vous alertent en cas de modification intempestive cassant le fonctionnement de l'ensemble ou induisant une régression ailleurs.

Le chapitre suivant sera consacré aux tests fonctionnels. Ils seront d'ailleurs abordés et implémentés dans les modules `job` et `category`. Prenez un peu de temps d'abord pour écrire quelques tests unitaires pour les classes de modèle de `Jobeet` ; cet exercice sera excellent pour vous préparer au chapitre suivant…

EN COULISSE

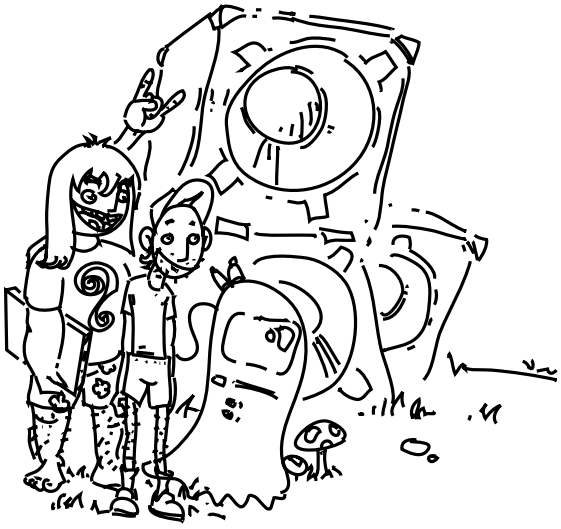
9 000 tests unitaires pour Symfony

Le framework Symfony lui-même possède plus de 9 000 tests unitaires à son actif pour en valider la fiabilité et la robustesse !


symfony

chapitre 9

```
~/work/jobeet $ ./symfony test:functional frontend categoryActions
# get /category/index
ok 1 - request parameter module is category
not ok 2 - request parameter action is index
#     Failed test (/Users/fabien/work/symfony/dev/1.2/lib/test/sfTesterRequest.class.php at line 48)
#            got: 'show'
#       expected: 'index'
not ok 3 - status code is 200
#     Failed test (/Users/fabien/work/symfony/dev/1.2/lib/test/sfTesterResponse.class.php at line 257)
#            got: 404
#       expected: 200
ok 4 - response selector body does not match regex /This is a temporary page/
1..4
 Looks like you failed 2 tests of 4.
~/work/jobeet $
```



Les tests fonctionnels

Parmi les autres types de tests automatisés figurent les tests fonctionnels. Ces derniers permettent de valider le comportement général des fonctionnalités d'une application en simulant la navigation d'un utilisateur dans son navigateur Internet. Symfony intègre par défaut un sous-framework de tests fonctionnels puissant qui simplifie l'écriture de scénarios de tests fonctionnels grâce à une interface fluide.

MOTS-CLÉS :

- Tests fonctionnels
- Objets sfBrowser et sfTestFunctional
- Testeurs Request et Response

Le chapitre précédent visait à expliquer comment tester unitairement les classes de `Jobeet` à partir de la librairie `lime` embarquée dans Symfony. Dans cette neuvième partie, ce sera l'occasion de découvrir et d'écrire des tests pour les fonctionnalités déjà implémentées dans les modules `job` et `category`.

Découvrir l'implémentation des tests fonctionnels

Les tests fonctionnels constituent un outil efficace pour tester une application du début à la fin, c'est-à-dire de la requête construite par le navigateur jusqu'à la réponse envoyée par le serveur. Leur champ d'action s'étend à toutes les couches de l'application comme le routage, le modèle, les actions et les templates. Cette section définit dans un premier temps la notion de tests fonctionnels puis explique comment ils sont intégrés au sein du framework Symfony.

En quoi consistent les tests fonctionnels ?

Concrètement, les tests fonctionnels ressemblent de très près aux tests manuels qu'un développeur réalise dans son navigateur. L'approche manuelle reste encore le schéma le plus couramment employé pour tester une application. Dès lors qu'une fonctionnalité est ajoutée ou modifiée, il est coutume d'ouvrir le navigateur pour vérifier que celle-ci se comporte correctement en contrôlant les différents objets qui composent la page de rendu final. Les liens, les images, les tableaux, les messages d'informations, etc., figurent parmi ces éléments qui servent de points de contrôle.

Toutefois, les tests manuels posent quelques problèmes car ils sont avant tout pénibles et fastidieux à réaliser. Il en résulte alors tout naturellement de multiples conséquences sur leur efficacité et leur viabilité. Au bout d'un certain temps passé à développer une application, le développeur aura tendance à se lasser et à survoler certaines fonctionnalités de l'application, d'où la nécessité d'automatiser la procédure de test. La question qui se pose alors est la suivante : « qu'est-ce qu'un test efficace et viable ? »

Un test viable, c'est avant tout un plan d'action que l'on doit être capable de répéter à l'identique et à l'infini. C'est donc là tout l'intérêt de faire appel aux services de la machine plutôt qu'aux compétences de l'être humain. En effet, ce dernier ne sera jamais en mesure de reproduire à l'identique et sur une période infinie le même test. Les tests fonctionnels sont donc apparus pour répondre à cette problématique, et c'est pour cette raison qu'ils s'intègrent parfaitement dans l'environnement de Symfony.

Implémentation des tests fonctionnels

Dans Symfony, les tests fonctionnels offrent une manière simple de décrire les scénarios de test. Chaque scénario peut être joué automatiquement à volonté en simulant l'expérience que possède un utilisateur dans son navigateur. Au même titre que les tests unitaires, ils apportent l'assurance que le code est fonctionnel et sans dysfonctionnements.

Il faut tout de même garder à l'esprit que le framework de tests fonctionnels ne remplace en aucun cas des outils tels que Selenium. Selenium s'exécute directement dans le navigateur afin d'automatiser les tests sur plusieurs plateformes, navigateurs et autres. Cet outil est également connu pour sa capacité à pouvoir tester le code Javascript d'une application.

Manipuler les composants de tests fonctionnels

Le framework interne de tests fonctionnels de Symfony fournit de nombreux composants pour faciliter l'écriture de tests de différentes natures. Le premier mis à l'étude dans ces prochaines lignes permet de simuler le comportement d'un navigateur web.

Simuler le navigateur grâce à l'objet sfBrowser

Les tests fonctionnels de Symfony s'exécutent à travers un navigateur un peu spécial implémenté par la classe `sfBrowser`. Celui-ci agit comme un navigateur taillé sur mesure pour l'application et directement connecté à celle-ci, sans nécessiter de serveur web.

Ce pseudo navigateur donne par exemple accès à tous les objets internes de Symfony avant et après chaque requête, ce qui offre l'opportunité de les introspecter et d'effectuer les vérifications souhaitées en cours d'exécution du programme.

Tester la navigation en simulant le comportement d'un véritable navigateur

La classe `sfBrowser` dispose d'un certain nombre de méthodes qui simulent la navigation comme le permet un navigateur web classique. Le tableau ci-dessous décrit celles qui sont principalement utilisées dans l'écriture de scénarios de tests.

Tableau 9–1 Liste des méthodes de l'objet sfBrowser

Nom de la méthode	Description
`get()`	Récupère une URL
`post()`	Poste des données vers une URL
`call()`	Appelle une URL (utilisé pour les méthodes `PUT` et `DELETE`)
`back()`	Retourne vers la page précédente de l'historique
`forward()`	Dirige vers la page suivante de l'historique
`reload()`	Recharge la page courante
`click()`	Clique sur un bouton ou un lien
`select()`	Sélectionne un bouton radio ou une case à cocher
`deselect()`	Désélectionne un bouton radio ou une case à cocher
`restart()`	Réinitialise le navigateur

Le code suivant illustre quelques exemples d'utilisation des méthodes de la classe `sfBrowser`.

```
$browser = new sfBrowser();

$browser->
  get('/')->
  click('Design')->
  get('/category/programming?page=2')->
  get('/category/programming', array('page' => 2))->
  post('search', array('keywords' => 'php'));
```

Modifier le comportement du simulateur de navigateur

La classe `sfBrowser` contient aussi quelques méthodes complémentaires qui offrent la possibilité de configurer le comportement du navigateur, comme le montre le tableau suivant.

Tableau 9–2 Liste des méthodes de l'objet sfBrowser qui permettent de configurer le comportement du navigateur

Nom de la méthode	Description
`setHttpHeader()`	Définit un en-tête HTTP
`setAuth()`	Définit les droits d'authentification de base
`setCookie()`	Fixe un cookie
`removeCookie()`	Retire un cookie
`clearCookies()`	Nettoie tous les cookies en cours
`followRedirect()`	Suit la redirection déclenchée

Pour fonctionner, les tests fonctionnels nécessitent l'utilisation d'un autre objet : l'objet `sfTestFunctional`. Cet dernier contient un ensemble de testeurs capables d'analyser les différents objets internes du framework comme la requête, la réponse, le routage, les formulaires et bien d'autres encore.

Préparer et exécuter des tests fonctionnels

La plupart des tests fonctionnels peuvent être réalisés à l'aide des objets `sfBrowser` et `lime`, et leurs méthodes respectives telles que `getRequest()` ou `getResponse()`. Néanmoins, l'idéal est de posséder un moyen d'instrospecter les objets internes de Symfony pour le scénario en cours. Heureusement, le framework fournit son lot de méthodes de test à l'aide de la classe `sfTestFunctional`. Le constructeur de cette dernière accepte une instance de la classe `sfBrowser` comme argument.

Comprendre la structure des fichiers de tests

L'objet `sfTestFunctional` délègue tous les tests à des objets « testeurs », dont la plupart sont embarqués par défaut dans Symfony. Chaque testeur est en réalité un objet qui étend la classe `sfTester`, ce qui permet par exemple de créer ses propres testeurs ou bien d'enrichir les existants en profitant de l'héritage de classes.

Dans un projet Symfony, tous les fichiers de tests fonctionnels se trouvent dans le répertoire `test/functional/`. Dans le cas présent de l'application développée, tous sont situés dans le sous-dossier `test/functional/frontend` puisque chaque application dispose de son propre répertoire. Celui-ci contient déjà deux fichiers qui ont été automatiquement générés lorsque les deux modules `job` et `category` ont été créés. Les fichiers `categoryActionsTest.php` et `jobActionsTest.php` renferment chacun quelques exemples de tests fonctionnels très basiques comme le présente le listing de code ci-après.

Mettre en place un jeu de tests fonctionnels en utilisant le chaînage de méthodes

Tests fonctionnels autogénérés pour le module category dans le fichier 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()
;
```

À première vue, ce script peut paraître étrange et peu commode pour la plupart des développeurs car sa syntaxe est peu singulière. Toutes les méthodes appelées sur l'objet `sfTestFunctional` (`$browser`) sont en effet chaînées afin d'assurer une interface fluide d'écriture de scénarios mais aussi une meilleure lisibilité du code.

Comment cette syntaxe est-elle rendue possible ? C'est techniquement enfantin puisque toutes les méthodes implémentées dans les classes `sfBrowser` et `sfTestFunctional` retournent toujours la référence à l'objet lui-même, conservée dans la variable `$this`.

Mettre en place un jeu de tests fonctionnels sans chaînage de méthodes

Le code ci-dessous est strictement identique au dernier avec le chaînage des méthodes en moins. Le résultat de la comparaison des deux syntaxes est clair : la première facilite grandement la lisibilité du code tandis que la seconde la réduit en raison de l'importance de bruit généré par la répétition de la variable `$browser`.

```
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();
```

Effectuer des tests dans le contexte d'un bloc testeur

Tous les tests sont exécutés dans le contexte d'un bloc testeur.

Le bloc d'un testeur commence par `with('TESTER NAME')->begin()` et s'achève par `end()` comme le montre l'exemple de code suivant.

```
$browser->
  with('request')->begin()->
    isParameter('module', 'category')->
    isParameter('action', 'index')->
  end();
```

Ce code vérifie si le paramètre `module` de la requête (testeur `request`) équivaut bien à la valeur `category`, et que le paramètre `action` contient lui aussi la valeur `index`. Lorsque l'on souhaite appeler seulement une méthode sur un testeur, il n'est pas nécessaire de créer un bloc de tests : `with('request')->isParameter('module', 'category')`.

Découvrir le testeur sfTesterRequest

Le testeur `sfTesterRequest` fournit des méthodes qui permettent d'introspecter et de tester les valeurs des propriétés de l'objet `sfWebRequest`. Le tableau ci-dessous présente quelques-unes d'entre elles.

Tableau 9–3 Liste des méthodes de l'objet sfTesterRequest

Nom de la méthode	Description
`isParameter()`	Contrôle la valeur d'un paramètre de la requête
`isFormat()`	Vérifie le format d'une requête
`isMethod()`	Vérifie la méthode (`GET`, `POST`, `PUT`, `DELETE`…)
`hasCookie()`	Indique si la requête a un cookie correspondant au nom donné en paramètre
`isCookie()`	Teste la valeur d'un cookie

Un testeur `sfTesterResponse` existe aussi afin de pouvoir contrôler les différents paramètres que le serveur envoie au client en guise de réponse à une requête HTTP.

Découvrir le testeur sfTesterResponse

Le testeur `sfTesterResponse` fournit quant à lui des méthodes qui permettent d'introspecter et de tester les valeurs des propriétés de l'objet `sfWebResponse`. Le tableau ci-dessous en présente quelques unes.

Tableau 9–4 Liste des méthodes de l'objet sfTesterResponse

Nom de la méthode	Description
`checkElement()`	Vérifie si un sélecteur CSS de la réponse correspond à un critère
`isHeader()`	Contrôle la valeur d'un en-tête
`isStatusCode()`	Contrôle le code de statut de la réponse (200, 301, 404, 500…)
`isRedirected()`	Vérifie si la réponse courante est redirigée

Exécuter les scénarios de tests fonctionnels

Les tests fonctionnels s'exécutent de la même manière que les tests unitaires vus au chapitre précédent. Il existe trois manières de lancer des tests fonctionnels :

- la première consiste à appeler directement le fichier PHP et de l'exécuter à l'aide du binaire PHP comme le présente le code suivant :

```
$ php test/functional/frontend/categoryActionsTest.php
```

- une commande Symfony permet de réaliser exactement la même chose en simplifiant la syntaxe :

```
$ php symfony test:functional frontend categoryActions
```

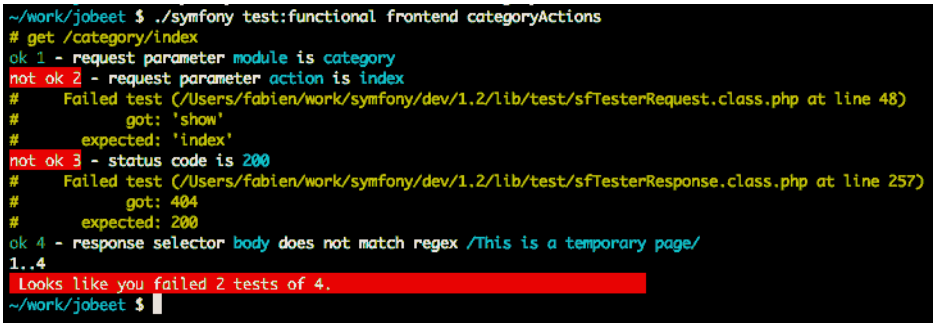


Figure 9–1
Résultat d'exécution des tests fonctionnels du module « category »

- enfin, comme pour les tests unitaires, la commande `test:functional` sert à lancer tous les tests fonctionnels d'une même application en omettant simplement le second argument :

```
$ php symfony test:functional frontend
```

Charger des jeux de données de tests

Au même titre que les tests unitaires pour le modèle de données Doctrine, des jeux de données de tests doivent être chargés en base de données chaque fois qu'un fichier de tests fonctionnels est exécuté. Le code écrit au chapitre précédent qui remplit cette tâche est réutilisable ici de la même manière.

```
include(dirname(__FILE__).'/../../bootstrap/functional.php');

$browser = new JobeetTestFunctional(new sfBrowser());
Doctrine::loadData(sfConfig::get('sf_test_dir').'/fixtures');
```

Charger des données dans un fichier de tests fonctionnels est un peu plus simple qu'avec les tests unitaires puisque la base de données est déjà initialisée par le script d'amorçage. Comme pour les tests unitaires, il est bien évidemment inutile de copier et coller ce bout de code dans chaque fichier de test. La meilleure manière de procéder est de mutualiser ce code dans une classe de tests fonctionnels dédiée qui hérite de `sfTestFunctional`.

Contenu du fichier lib/test/JobeetTestFunctional.class.php

```
class JobeetTestFunctional extends sfTestFunctional
{
  public function loadData()
  {
    Doctrine::loadData(sfConfig::get('sf_test_dir').'/fixtures');

    return $this;
  }
}
```

Écrire des tests fonctionnels pour le module d'offres

Écrire des tests fonctionnels revient exactement à jouer un scénario dans un navigateur web. Il s'agit en effet de tester que chaque fonctionnalité testée réagit comme le définit son cahier des charges. De plus, les tests fonctionnels ont pour objectifs de vérifier que le rendu final de chaque page correspond exactement à ce que le développeur a prévu dans son code.

Le second chapitre de cet ouvrage décrit de manière non exhaustive tous les besoins fonctionnels de l'application. En y réfléchissant bien, ces cas d'utilisation sont exactement la description littérale des scénarios de tests fonctionnels à écrire. Pour la page d'accueil du module d'offres d'emploi, on ne compte pas moins de cinq tests obligatoires qui seront développés juste après :

- les offres d'emploi expirées ne sont plus affichées ;
- seulement N offres d'emploi actives sont listées par catégorie ;
- une catégorie possède un lien vers sa page dédiée s'il y a trop d'offres d'emploi ;
- les offres d'emploi sont listées par date ;
- chaque offre d'emploi de la page d'accueil est cliquable.

Il est temps de démarrer avec le premier scénario de cette liste.

Les offres d'emploi expirées ne sont pas affichées

Ce test fonctionnel tient en quelques lignes de code PHP puisqu'il s'agit tout simplement de vérifier que la page d'accueil ne contient aucune occurrence du sélecteur CSS 3 passé en paramètre de la méthode `checkElement()` du testeur `response`.

Contenu du fichier 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 inséré en appelant la méthode `info()` dans le but de rendre la sortie plus lisible et compréhensible. Le contrôle de l'exclusion des offres d'emploi de la page d'accueil se traduit par le fait que le sélecteur CSS 3 `.jobs td.position:contains("expired")` ne doit en aucun cas être présent dans le contenu HTML de la réponse.

Si l'on se souvient des fichiers de données de tests, la seule offre d'emploi expirée contient la chaîne « expired » en guise de poste. Lorsque le second argument de la méthode `checkElement()` est un booléen, la méthode teste l'existence des nœuds qui correspondent au sélecteur CSS 3. La méthode `checkElement()` supporte d'ailleurs la plupart des sélecteurs CSS 3 existants.

Seulement N offres sont listées par catégorie

Tester qu'une catégorie présente bien un certain nombre fini d'offres est relativement simple avec les sélecteurs CSS et la méthode `checkElement()`. En effet, il s'agit ici de compter le nombre de lignes du tableau HTML pour une même catégorie. Pour faciliter davantage le test, il suffit de s'appuyer sur le nom de la catégorie qui est généré dans le code HTML sous forme d'une classe CSS.

Suite du contenu du fichier 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()` est également capable de vérifier qu'un sélecteur CSS correspond à un nombre fini de nœuds dans le document, en passant un entier comme second argument.

Un lien vers la page d'une catégorie est présent lorsqu'il y a trop d'offres

Dans Jobeet, lorsqu'une catégorie regroupe sur la page d'accueil plus d'offres d'emploi que le nombre maximum d'offres autorisé pour cette dernière, un lien « more jobs » est affiché. L'objectif de ce test fonctionnel consiste à vérifier la présence ou l'absence du lien en fonction du nombre d'offres d'emploi dans chaque catégorie. D'après les fichiers de données de tests, seule la catégorie « Programming » est censée accueillir ledit lien.

Suite du fichier 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()
;
```

Ces tests vérifient qu'il n'y a pas de lien « more jobs » pour la catégorie « design », c'est-à-dire que le sélecteur CSS `.category_design .more_jobs` existe nulle part dans le document. Ils testent en revanche que le lien « more jobs » est bien présent pour la catégorie « programming », donc que le sélecteur CSS `.category_programming .more_jobs` existe.

Les offres d'emploi sont triées par date

Ce test est un peu plus complexe à mettre en œuvre que les précédents… En effet, pour tester si les offres d'emploi sont effectivement ordonnées par date, il faut vérifier que la première offre d'emploi qui apparaît dans la page d'accueil correspond bien à celle que l'on attend. Ce résultat peut être obtenu en contrôlant que l'URL contient la clé primaire attendue. Or, les clés primaires ne sont pas fixes et peuvent donc varier entre deux exécutions d'un même fichier de tests fonctionnels. Cela est dû au fait que Doctrine recharge les données de test dans la base de données à chaque nouvelle exécution du script. L'astuce pour y parvenir est en fait triviale puisqu'il s'agit de récupérer l'objet Doctrine de la base de données comme le montre le code suivant.

```
$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()
;
```

Quelques explications s'imposent malgré tout dans la mesure où le code présenté se révèle un peu complexe. Tout d'abord, la requête Doctrine récupère l'offre d'emploi la plus récente pour la catégorie dont le slug a pour valeur « programming ».

Puis, la méthode `checkElement()` teste la présence du sélecteur CSS 3 passé en paramètre. Ce dernier est formaté par la fonction `sprintf()` de manière à signifier que la première offre d'emploi (`tr:first`) de la catégorie « programming » (`.category_programming`) possède un lien dont l'attribut `href` contienne la clé primaire de l'objet attendu (`a[href*="/%d/"]`).

Bien que ce test fonctionne parfaitement, un besoin de remaniement se fait ressentir. En effet, la récupération de la première offre d'emploi de la catégorie « programming » est potentiellement réutilisable ailleurs dans les tests. Le code ne peut en revanche être déplacé vers la couche du modèle dans la mesure où il est purement spécifique aux tests. De ce

postulat, on en déduit clairement que la meilleure place à lui consacrer est la classe `JobeetTestFunctional` créée plus tôt dans ce chapitre. Celle-ci se comporte comme une classe de tests fonctionnels propre à l'environnement de test.

Méthode à ajouter au fichier 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');
    $q = Doctrine::getTable('JobeetJob')-
>addActiveJobsQuery($q);

    return $q->fetchOne();
  }

  // ...
}
```

Le code des tests fonctionnels précédent peut alors être réduit à celui qui suit.

Suite du fichier 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()
;
```

Chacune des offres de la page d'accueil est cliquable

Pour tester le lien d'une offre de la page d'accueil, il suffit de simuler un clic sur le texte « Web Developper ». Comme il y en a plusieurs possibles sur cette page, le test force explicitement le navigateur à cliquer sur le premier qu'il trouve (`array('position' => 1)`).

Chaque paramètre de la requête est ensuite testé pour s'assurer que le routage a correctement fait son travail.

```
$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', 'sensio-labs')->
    isParameter('location_slug', 'paris-france')->
    isParameter('position_slug', 'web-developer')->
    isParameter('id', $browser->getMostRecentProgrammingJob()-
>getId())->
  end()
;
```

Autres exemples de scénarios de tests pour les pages des modules job et category

Cette section fournit tout le code nécessaire pour tester les pages des modules `job` et `category`. En lisant le code avec attention, on apprend quelques nouvelles astuces pratiques.

Contenu du fichier lib/test/JobeetTestFunctional.class.php

```
class JobeetTestFunctional extends sfTestFunctional
{
  public function loadData()
  {
    Doctrine::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');
    $q = Doctrine::getTable('JobeetJob')->addActiveJobsQuery($q);

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

```
  // Récupération d'une offre expirée
  public function getExpiredJob()
  {
    $q = Doctrine_Query::create()
      ->from('JobeetJob j')
      ->where('j.expires_at < ?', date('Y-m-d', time()));

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

Contenu du fichier 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()
;

$browser->info('2 - The job page')->
  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', 'sensio-labs')->
    isParameter('location_slug', 'paris-france')->
    isParameter('position_slug', 'web-developer')->
    isParameter('id', $browser->getMostRecentProgrammingJob()->
                        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)
;
```

Contenu du fichier 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('22')->
  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#')
;
```

Déboguer les tests fonctionnels

Il arrive parfois qu'un test fonctionnel échoue. Comme Symfony simule un navigateur sans interface graphique, il peut devenir difficile de diagnostiquer rapidement l'origine du problème. Heureusement, le framework fournit la méthode `debug()` pour imprimer le contenu et l'entête de la réponse sur la sortie standard.

```
$browser->with('response')->debug();
```

La méthode `debug()` peut être insérée n'importe où dans le bloc du testeur de la réponse. Son appel forcera l'arrêt immédiat de l'exécution du script.

Exécuter successivement des tests fonctionnels

La tâche `test:functional` peut aussi être utilisée pour lancer tous les tests fonctionnels d'une même application.

```
$ php symfony test:functional frontend
```

La tâche affiche en sortie une ligne de résultat pour chaque fichier testé.

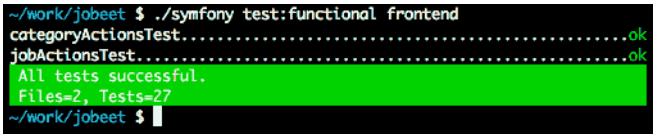


Figure 9–2
Résultat d'exécution des tests fonctionnels de l'application « frontend »

Exécuter les tests unitaires et fonctionnels

Il est bien évidemment possible d'exécuter tous les tests unitaires et fonctionnels d'une application. C'est en effet la tâche `test:all` qui a le rôle de lancer à la suite tous les fichiers de tests et de générer en sortie un rapport pour chacun d'eux.

```
$ php symfony test:all
```

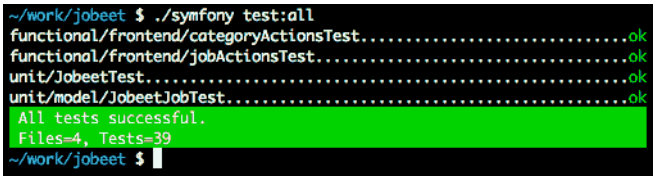


Figure 9–3
Résultat d'exécution de tous les tests de Jobeet

En résumé…

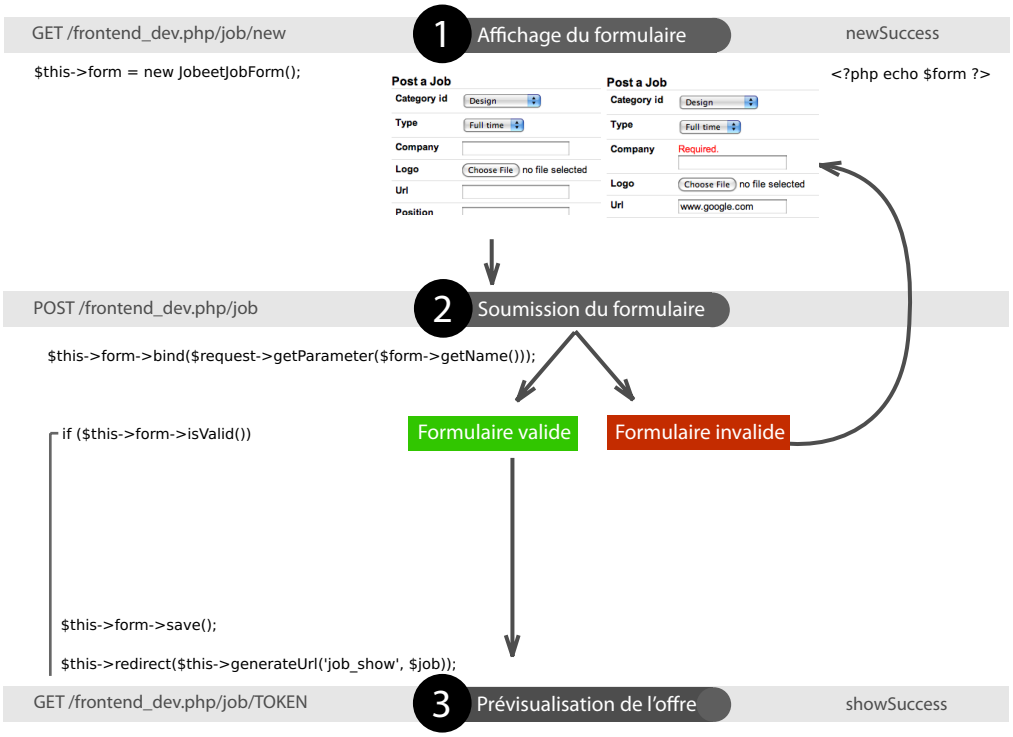
Ce chapitre achève le tour de présentation des outils de tests de Symfony. Désormais, il sera délicat de trouver un prétexte pour ne pas tester votre application ! Grâce à lime et au framework de tests fonctionnels, Symfony apporte des outils puissants pour réaliser l'écriture de tests avec peu d'effort.

Pour l'instant, les tests fonctionnels ont été relativement survolés, c'est pourquoi de nouveaux seront écrits au cours du projet à chaque fois que de nouvelles fonctionnalités seront implémentées ; ils seront l'occasion de découvrir de nouveaux atouts du framework de test.

Le chapitre suivant aborde une fonctionnalité essentielle de Symfony : le framework de formulaires…


symfony

chapitre 10


GET /frontend_dev.php/job/new
1
Affichage du formulaire
newSuccess
$this->form = new JobeetJobForm();
<?php echo $form ?>
Post a Job
Category id
Design
Type
Full time
Company
Logo
Choose File
no file selected
Url
Position
Post a Job
Category id
Design
Type
Full time
Company
Required.
Logo
Choose File
no file selected
Url
www.google.com
POST /frontend_dev.php/job
2
Soumission du formulaire
$this->form->bind($request->getParameter($form->getName()));
if ($this->form->isValid())
Formulaire valide
Formulaire invalide
$this->form->save();
$this->redirect($this->generateUrl('job_show', $job));
GET /frontend_dev.php/job/TOKEN
3
Prévisualisation de l'offre
showSuccess

Accélérer la gestion des formulaires

MOTS-CLÉS :

- Formulaires
- Validation des données utilisateurs
- Sécurité contre les failles CSRF

Les formulaires de saisie sont depuis toujours les principaux outils d'interaction avec l'utilisateur dans le but de récolter des informations. Bien qu'ils soient largement connus des développeurs, leur manipulation n'en reste pas moins difficile et est souvent à l'origine de nombreuses failles de sécurité du système d'information.

Ce chapitre montre comment Symfony simplifie grandement tant la création et la validation des formulaires que leur traitement automatique.

Ce dixième chapitre aborde l'une des principales grandes nouveautés de Symfony, apparue depuis la version 1.1 et enrichie dans cette nouvelle version. Il s'agit en effet du framework interne de création et de validation des formulaires. Celui-ci simplifie la gestion des formulaires web en remplissant lui-même les tâches de génération, de contrôle d'intégrité des données, de traitement automatique de ces dernières lorsqu'elles sont validées, et de sécurité.

À la découverte des formulaires avec Symfony

N'importe quel site Internet possède des formulaires. Cela va bien sûr du simple formulaire de contact aux plus complexes composés d'une pléiade de champs divers et variés. Pour un développeur, écrire des formulaires est aussi l'une des tâches les plus rébarbatives, les plus ardues et aussi les plus fastidieuses. Cette tâche nécessite en effet d'écrire le code HTML du formulaire, puis d'implémenter les règles de validation pour chaque champ, de traiter les valeurs pour les sauvegarder en base de données, d'afficher les messages d'erreur, de repeupler le formulaire en cas d'erreur, et bien plus encore…

Bien évidemment, au lieu de réinventer la roue à chaque fois, Symfony fournit un framework dédié aux formulaires afin d'en simplifier leur gestion. Celui-ci se compose en trois parties distinctes :

- la **validation** : le sous-framework de validation contient toutes les classes nécessaires à la validation des entrées (entiers, chaînes de caractères, adresses e-mail, dates…) ;
- les **widgets** : le sous-framework de widgets possède quant à lui les classes utiles à la génération du code HTML de chaque champ du formulaire (input, textarea, select…) ;
- les **formulaires** : les classes du sous-framework de formulaires représentent les formulaires conçus à partir des widgets et des validateurs, et fournissent les méthodes pour faciliter leur gestion. Chaque champ du formulaire dispose de son propre widget et de son ou ses propres validateurs.

Les formulaires de base

Dans Symfony, un formulaire n'est ni plus ni moins qu'une classe dans laquelle sont déclarés les champs. Chaque champ possède un nom, un widget et un ou plusieurs validateurs. Ainsi, un simple formulaire de contact peut être défini d'après la classe `ContactForm` suivante.

```
<?php

class ContactForm extends sfForm
{
  public function configure()
  {
    $this->setWidgets(array(
      'email'   => new sfWidgetFormInput(),
      'message' => new sfWidgetFormTextarea(),
    ));

    $this->setValidators(array(
      'email'   => new sfValidatorEmail(),
      'message' => new sfValidatorString(array('max_length' =>
255)),
    ));
  }
}
```

Les champs du formulaire sont tous déclarés dans la méthode `configure()` de la classe au moyen des méthodes `setWidgets()` et `setValidators()`.

Le framework de formulaires est par défaut livré avec un certain nombre de widgets et de validateurs prêts à l'emploi. L'API les décrit tous très largement en indiquant pour chacun toutes les options, erreurs et messages d'erreur par défaut.

Les noms des classes de widgets et de validateurs sont eux aussi très explicites. Le champ `email` sera rendu par une balise HTML `<input>` (`sfWidgetFormInput`) et validé comme une adresse e-mail (`sfValidatorEmail`). Le champ `message` quant à lui sera rendu sous la forme d'une balise HTML `<textarea>` (`sfWidgetFormTextarea`). Sa valeur doit être une chaîne de caractères d'une longueur strictement inférieure ou égale à 255 caractères. Par défaut, tous les champs sont considérés comme obligatoires car la valeur par défaut de l'option `required` est `true`. Ainsi, la définition de la règle de validation du champ `email` est équivalente à `new sfValidatorEmail(array('required' => true))`.

Un formulaire peut également être fusionné avec un autre en utilisant la méthode `mergeForm()`, ou bien embarquer un autre formulaire imbriqué grâce à la méthode `embedForm()`.

```
$this->mergeForm(new AnotherForm());
$this->embedForm('name', new AnotherForm());
```

Les formulaires générés par les tâches Doctrine

La plupart du temps, un formulaire a pour vocation d'être sérialisé en base de données. Symfony connaît déjà tout du modèle de base de données, ce qui lui permet d'automatiser la génération de formulaires basés sur ces informations. En fait, lorsque la tâche `doctrine:build-all` a été exécutée au chapitre 3, Symfony a automatiquement fait appel à la tâche `doctrine:build-forms`.

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

La tâche `doctrine:build-forms` génère les classes de modèle des formulaires dans le répertoire `lib/form/`. L'organisation de ces fichiers générés est semblable à celle de `lib/model/`. Chaque classe de modèle possède une classe de formulaire associée. Cette dernière est vide par défaut puisqu'elle hérite des propriétés et des méthodes de la superclasse de base qui contient l'ensemble de la configuration du formulaire. La classe `JobeetJobForm` est un exemple de formulaire Doctrine lié au modèle `JobeetJob`.

Contenu du fichier lib/form/doctrine/JobeetJobForm.class.php

```
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
  }
}
```

C'est en parcourant les fichiers générés du répertoire `lib/form/doctrine/base/` que l'on découvre des exemples étonnants d'usage des widgets et des validateurs natifs de Symfony.

Personnaliser le formulaire d'ajout ou de modification d'une offre

Le formulaire des offres est l'exemple parfait pour apprendre à personnaliser des classes de formulaires. Afin de garantir une meilleure compréhension du processus de personnalisation, ce dernier sera présenté pas à pas dans les prochaines sections. La première étape consiste à changer le lien « Post a job » du layout afin de contrôler les modifications directement dans le navigateur.

Code à placer dans le fichier apps/frontend/templates/layout.php

```
<a href="<?php echo url_for('@job_new') ?>">Post a Job</a>
```

L'objectif suivant présente la manière de retirer des champs d'un formulaire autogénéré.

Supprimer les champs inutiles du formulaire généré

Par défaut, les formulaires Doctrine affichent tous les champs pour chaque colonne d'une table de la base de données. Or, pour le formulaire de création d'offre d'emploi, certaines valeurs de la table `jobeet_job` ne doivent pas être éditées par l'utilisateur final. Il faut donc retirer du formulaire leur champ associé. La manipulation est triviale : Il s'agit tout simplement de supprimer ces champs directement dans la méthode `configure()` du formulaire comme le montre l'exemple ci-dessous.

Contenu du fichier 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']
    );
  }
}
```

Supprimer un champ avec `unset()` revient à supprimer aussi bien son widget que son validateur. La syntaxe présentée ci-dessus peut paraître exotique et surprenante à première vue. Comment un objet peut-il se comporter comme un tableau ? La réponse se trouve dans la déclaration de la classe `sfForm`. Celle-ci implémente effectivement l'interface `ArrayAccess` de la SPL de PHP 5, en redéfinissant les quatre méthodes de cette dernière. Cette syntaxe a l'avantage d'être à la fois plus simple et plus familière pour les développeurs PHP.

Redéfinir plus précisément la configuration d'un champ

La configuration d'un formulaire doit parfois être plus précise que celle qui a été générée par l'analyse interne du modèle de la base de données. Par exemple, la colonne `email` est déclarée comme étant un `varchar` dans le schéma, mais sa valeur doit, comme son nom l'indique, être validée comme une véritable adresse de courrier électronique. Il en va de même pour le type d'offre. Ce champ est défini comme un `varchar`, mais dans la réalité, son widget associé se présentera sous la forme d'une liste déroulante de choix prédéfinis.

Utiliser le validateur sfValidatorEmail

La brève introduction précédente explique clairement que la valeur du champ `email` doit être acceptée uniquement si son format correspond à celui d'une adresse électronique valide. Heureusement, Symfony fournit un validateur prêt à l'emploi pour remplir cette tâche. Il suffit donc simplement de surcharger la configuration du champ `email` en lui appliquant ce nouveau validateur `sfValidatorEmail`.

Définition du validateur sfValidatorEmail pour le champ email dans le fichier lib/form/doctrine/JobeetJobForm.class.php

```
public function configure()
{
  // ...

  $this->validatorSchema['email'] = new sfValidatorEmail();
}
```

Remplacer le champ permettant le choix du type d'offre par une liste déroulante

Bien que le type de la colonne `type` de la table `jobeet_job` est un `varchar`, la valeur de ce champ doit être restreinte à une liste prédéfinie de choix : « full time », « part time » ou bien « freelance ». Le widget le mieux adapté pour ce cas de figure est sans aucun doute une liste déroulante identifiée par une balise HTML `<select>`. Une liste de boutons radio ferait également très bien l'affaire, du fait qu'il y a peu de choix possibles.

Définir la liste des valeurs autorisées

La première étape consiste tout d'abord à définir la liste des choix possibles sous la forme d'un simple tableau associatif PHP. L'endroit le plus approprié pour établir celle-ci n'est pas forcément la classe du formulaire mais le modèle, et plus exactement la classe `JobeetJobTable` comme le montre l'exemple suivant.

Déclaration de la liste des types de poste dans le fichier 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;
  }

  // ...
}
```

Le tableau associatif `$types` possède en clé la valeur à sauvegarder en base de données, donc la valeur de chaque nœud `<option>` de la liste déroulante, associée à la chaîne à afficher en guise de label dans chaque balise `<option>`.

Implémenter le widget sfWidgetFormChoice

La seconde étape de personnalisation du champ `type` consiste maintenant à instancier un objet `sfWidgetFormChoice`, auquel est attribué la liste des types prédéfinis pour remplir la liste déroulante.

```
$this->widgetSchema['type'] = new sfWidgetFormChoice(array(
  'choices' => Doctrine::getTable('JobeetJob')->getTypes(),
  'expanded' => true,
));
```

`sfWidgetFormChoice` représente un widget de choix qui peut être rendu par un autre widget différent selon la définition de ses options de configuration (`expanded` ou `multiple`). La liste ci-dessous fait le bilan de toutes les possibilités de configuration de ce widget, et la forme qu'il prendra lorsqu'il sera rendu dans sa version HTML.

- Liste déroulante (`<select>`) : `array('multiple' => false, 'expanded' => false)`
- Liste déroulante multiple (`<select multiple="multiple">`) : `array('multiple' => true, 'expanded' => false)`
- Liste de boutons radio : `array('multiple' => false, 'expanded' => true)`
- Liste de cases à cocher : `array('multiple' => true, 'expanded' => true)`

Pour forcer la sélection d'un bouton radio par défaut (`full-time` par exemple), il suffit de changer la valeur par défaut dans le schéma de description de la base de données.

Valider la valeur choisie par l'utilisateur avec sfValidatorChoice

La dernière étape de personnalisation concerne désormais la validation de la donnée transmise par ce champ. Contrairement à ce qu'on pourrait croire, le fait d'utiliser une liste déroulante pour conditionner les choix possibles ne garantit en rien la fiabilité de la valeur soumise. En effet,

n'importe qui peut soumettre une valeur non valide. Un hacker, par exemple, outrepassera facilement la liste déroulante en utilisant des outils comme Curl (Client URL Request Library, permettant de récupérer le contenu d'une ressource accessible à une URL) ou la célèbre Firefox Web Developper Toolbar. Le code qui suit ajoute un validateur `sfValidatorChoice` qui permet de contrôler que la valeur saisie correspond bien à l'une des valeurs autorisées du widget.

```
$this->validatorSchema['type'] = new sfValidatorChoice(array(
  'choices' => array_keys(Doctrine::getTable('JobeetJob')-
>getTypes()),
));
```

La configuration du validateur est très aisée puisqu'il s'agit de fournir une liste des valeurs autorisées à l'option `choices` de ce dernier. C'est exactement ce que réalise la fonction `array_keys()` de PHP qui retourne un tableau classique dont les valeurs sont cette fois-ci les clés du tableau `$types` définis plus haut.

Personnaliser le widget permettant l'envoi du logo associé à une offre

Le champ logo stocke la valeur du nom de fichier du logo associé à l'offre. Or, pour permettre à l'utilisateur d'ajouter un logo à son offre, le widget du champ logo doit impérativement être transformé en un champ `input` de type `file`. Bien sûr, chaque fichier transmis doit également être contrôlé pour éviter le téléchargement de documents potentiellement dangereux pour l'application. De ce fait, le fichier téléchargé devra impérativement être une image.

Implémenter le widget sfWidgetFormInputFile

Dans Symfony, un champ de téléchargement de fichier est déclaré au moyen du widget `sfWidgetFormInputFile`. Ce dernier retourne le code HTML d'un champ `<input>` de type `file` qui permet à l'utilisateur de proposer des fichiers à télécharger depuis le disque dur de son ordinateur.

```
$this->widgetSchema['logo'] = new sfWidgetFormInputFile(array(
  'label' => 'Company logo',
));
```

Il ne reste enfin qu'à spécifier un validateur pour contrôler automatiquement les caractéristiques du fichier transmis. C'est le rôle du validateur `sfValidatorFile`.

Valider les fichiers avec sfValidatorFile

En proposant à l'utilisateur d'envoyer un fichier pour illustrer son offre, c'est aussi là une ouverture du serveur aux fichiers potentiellement malveillants. C'est pour cette raison qu'il faut s'assurer que chaque fichier transmis est sain et ne comporte pas de risque pour le système d'information qui l'accueille. Dans le cas de Jobeet, les fichiers autorisés sont uniquement des images, ce qui limite considérablement les risques d'infection du système. Néanmoins, les images ont des caractéristiques propres, comme l'extension ou le type de contenu, qui sont faciles à valider. Pour ce faire, Symfony met à disposition le validateur `sfValidatorFile` qui contient par défaut une configuration spécifique pour la validation des images.

```
$this->validatorSchema['logo'] = new sfValidatorFile(array(
  'required'   => false,
  'path'       => sfConfig::get('sf_upload_dir').'/jobs',
  'mime_types' => 'web_images',
));
```

Le code ci-dessus présente une option `mime_types` avec la valeur `web_images`. Il s'agit en fait de la définition de la configuration du validateur pour le contrôle des fichiers d'images.

Que se passe-t-il concrètement lorsqu'un utilisateur soumet un fichier depuis le formulaire ? Tout d'abord, le validateur `sfValidatorFile` s'assure que le fichier transmis est bien une image au format web. Il le renomme ensuite avec un nom arbitraire et unique afin de supprimer les espaces et autres caractères accentués. Puis, il copie ce fichier dans le répertoire défini par l'option `path` avant de mettre finalement à jour la valeur de la colonne `logo` avec le nouveau nom du fichier.

REMARQUE Création du répertoire d'upload des logos

La création du répertoire `web/uploads/jobs/` n'est pas gérée par Symfony. Cela doit être fait manuellement en s'assurant que le répertoire possède les droits d'écriture nécessaires.

Implémenter l'affichage du logo dans le template

Sachant que le validateur sauvegarde le chemin relatif dans la base de données, le chemin utilisé dans le template `showSuccess.php` doit être édité en conséquence.

Code à remplacer dans le template apps/frontend/modules/job/template/showSuccess.php

```
<img src="/uploads/jobs/<?php echo $job->getLogo() ?>"
alt="<?php echo $job->getCompany() ?> logo" />
```

Si une méthode `generateLogoFilename()` existe dans le modèle, elle sera automatiquement appelée par le validateur afin de redéfinir le processus natif de génération du nom de fichier. Cette méthode prend un objet de type `sfValidatedFile` comme argument.

Modifier plusieurs labels en une seule passe

Symfony attribue par défaut un label (balise HTML `<label>`) spécifique à chaque champ du formulaire, en se basant sur leur nom respectif. Bien évidemment, chaque valeur de label est entièrement personnalisable, et peut être spécifiée soit dans la configuration du widget grâce à l'option `label`, soit directement et pour plusieurs champs à la fois grâce à la méthode `setLabels()`. La première méthode a été présentée lors de la configuration du widget du champ `logo` quelques lignes plus haut. La seconde quant à elle est expliquée dans le code ci-dessous.

```
$this->widgetSchema->setLabels(array(
  'category_id'    => 'Category',
  'is_public'      => 'Public?',
  'how_to_apply'   => 'How to apply?',
));
```

La méthode `setLabels()` prend un tableau associatif en paramètre dont la clé correspond au nom du champ, et la valeur à la chaîne qui doit être affichée dans la balise `<label>` générée.

Ajouter une aide contextuelle sur un champ

De la même manière que le label d'un champ peut être surchargé, Symfony propose un moyen d'ajouter une aide contextuelle aux champs du formulaire. L'objectif de cette dernière est d'apporter une information à la fois plus significative et plus pertinente que le label du champ lui-même. Par exemple, le champ `is_public` du formulaire en nécessite une.

```
$this->widgetSchema->setHelp('is_public', 'Whether the job can
also be published on affiliate websites or not.');
```

Dans ce cas précis, cette aide indique à l'utilisateur s'il souhaite publier ou non son offre sur les sites web partenaires pour qu'elle ait plus de visibilité.

Présentation de la classe finale de configuration du formulaire d'ajout d'une offre

Après toutes les modifications apportées dans ces dernières sections, la classe `JobeetJobForm` est désormais la suivante.

Contenu du fichier lib/form/doctrine/JobeetJobForm.class.php

```
<?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 sfValidatorEmail();

    $this->widgetSchema['type'] = new sfWidgetFormChoice(array(
      'choices' => Doctrine::getTable('JobeetJob')->getTypes(),
      'expanded' => true,
    ));
    $this->validatorSchema['type'] = new
sfValidatorChoice(array(
      'choices' => array_keys(Doctrine::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.');
  }
}
```

Il est temps de découvrir comment sont rendus les formulaires dans les templates, et de quelle manière leur habillage peut être personnalisé pour répondre aux besoins de la maquette graphique de l'application.

Manipuler les formulaires directement dans les templates

Générer le rendu d'un formulaire

Maintenant que la classe du formulaire est personnalisée, il ne reste plus qu'à l'afficher. Dans Jobeet, le template du formulaire est exactement le même, que l'on veuille créer une nouvelle offre ou que l'on souhaite en éditer une existante. En fait, les deux fichiers `newSuccess.php` et `editSuccess.php` sont sensiblement similaires.

Extrait du fichier apps/frontend/modules/job/templates/newSuccess.php

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

<h1>Post a Job</h1>

<?php include_partial('form', array('form' => $form)) ?>
```

Le formulaire est lui-même rendu dans le template partiel `_form.php`. L'application Jobeet nécessite un rendu légèrement différent pour ce formulaire, c'est pourquoi le contenu du fichier `_form.php` doit être remplacé par le code suivant.

Contenu du fichier apps/frontend/modules/job/templates/_form.php

```
<?php include_stylesheets_for_form($form) ?>
<?php include_javascripts_for_form($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 `include_stylesheets_for_form()` et `include_javascripts_for_form()` importent les fichiers CSS et JavaScript nécessaires au bon fonctionnement de certains widgets du formulaire. Bien que le formulaire d'offre ne nécessite ni CSS ni JavaScript particulier, la bonne pratique consiste à toujours conserver ces helpers

REMARQUE La feuille de style des offres

Si la feuille de style `job.css` n'a pas encore été ajoutée aux templates `newSuccess.php` et `editSuccess.php`, c'est le moment de le faire pour ces derniers en utilisant `<?php use_stylesheet('job.css') ?>`

dans le template à titre préventif. Ce sera, en effet, un réel gain de temps le jour où de véritables widgets spécifiques seront intégrés au formulaire. L'approche pragmatique reste bien souvent la meilleure !

Le helper `form_tag_for()` se charge de générer la balise `<form>` pour le formulaire et la route donnés. D'autre part, il fixe la méthode à `POST` ou `PUT` selon que l'objet associé au formulaire est nouveau ou non. Enfin, il prend garde à bien implémenter l'attribut spécial `multipart` si le formulaire contient au moins un widget de transfert de fichier. Dans le code précédent, le formulaire est entièrement rendu grâce à l'instruction `<?php echo $form ?>`. La structure du langage `echo` appelle implicitement la méthode `__toString()` du formulaire qui génère tous les widgets, labels, messages d'erreur et autres informations d'aide déclarés dans la classe `JobeetJobForm`.

Cependant, la plupart des formulaires que l'on trouve sur l'Internet sont très spécifiques et peuvent être, pour certains, relativement complexes à mettre en page. La section suivante apporte la documentation de base pour générer un formulaire Symfony à la main, champ par champ.

Personnaliser le rendu des formulaires

Par défaut, l'instruction `<?php echo $form ?>` génère le formulaire sous la forme d'un tableau HTML où chaque ligne correspond à un widget. Or, la plupart du temps, le layout d'un formulaire se révèle différent et plus complexe selon l'aspect général souhaité. Par exemple, il sera parfois nécessaire d'ajouter des balises `<fieldset>` pour séparer les blocs de même nature ou bien simplement pour afficher deux champs particuliers l'un à côté de l'autre.

Découvrir les méthodes de l'objet sfForm

L'objet de formulaire fournit plusieurs méthodes utiles pour faciliter la personnalisation du rendu afin de ne pas être contraint uniquement à une structure en tableau. Le tableau ci-dessous résume les principales méthodes qu'il est possible d'utiliser.

Tableau 10–1 Liste des méthodes utiles au rendu de l'objet sfForm

Méthode	Description
`render()`	Génère le formulaire (équivalent pour `echo $form`)
`renderHiddenFields()`	Génère les champs cachés
`hasErrors()`	Retourne `true` si le formulaire a des erreurs
`hasGlobalErrors()`	Retourne `true` si le formulaire a des erreurs globales
`getGlobalErrors()`	Retourne un tableau d'erreurs globales
`renderGlobalErrors()`	Génère les erreurs globales

Comprendre et implémenter les méthodes de l'objet sfFormField

Au même titre que pour l'objet `sfForm`, chaque champ peut être rendu individuellement grâce aux méthodes de la classe `sfFormField`. Dans un template, n'importe quel champ du formulaire peut être récupéré unitairement grâce à la syntaxe `ArrayAccess` de l'objet `sfForm`. Par exemple, `$form['company']` retourne l'objet `sfFormField` correspondant au champ `company` du formulaire. Le tableau qui suit dresse une liste des méthodes de rendu applicables à n'importe quel champ d'un formulaire.

Tableau 10–2 Liste des méthodes utiles au rendu de l'objet sfFormField

Méthode	Description
`renderRow()`	Génère la ligne complète du champ
`render()`	Retourne le code HTML du widget
`renderLabel()`	Retourne le code HTML de la balise `<label>`
`renderError()`	Génère le message d'erreur du champ
`renderHelp()`	Génère l'aide du champ

L'exemple de code suivant est une syntaxe équivalente à `echo $form`.

```
<?php foreach ($form as $widget): ?>
  <?php echo $widget->renderRow() ?>
<?php endforeach; ?>
```

Cet autre bout de code illustre la manière de générer individuellement un champ de formulaire complet dans un template au moyen des méthodes de l'objet `sfFormField`.

```
<div class="form_field">
  <?php echo $form['company']->renderError(); ?>
  <?php echo $form['company']->renderLabel(); ?>
  <?php echo $form['company']->render(); ?>
  <div class="form_field_help">
    <?php echo $form['company']->renderHelp(); ?>
  </div>
</div>
```

Manipuler les formulaires dans les actions

Les parties précédentes ont montré comment, dans Symfony, un formulaire est déclaré, puis configuré et enfin rendu dans un template. La dernière étape consiste donc à expliquer de quelle manière un formulaire est rendu dynamique dans les actions d'un module.

Découvrir les méthodes autogénérées du module job utilisant les formulaires

Pour l'instant, Jobeet dispose d'un formulaire capable de gérer la création et l'édition d'une offre. Ce dernier se résume toujours à une classe de configuration des champs ainsi qu'à un template partiel pour l'afficher. Il est désormais temps de le mettre en œuvre au sein des actions.

Le formulaire des offres est actuellement géré par cinq méthodes des actions du module `job` :

- **new** : affiche un formulaire vide pour créer une nouvelle offre ;
- **edit** : affiche un formulaire prérempli pour éditer une offre existante ;
- **create** : crée la nouvelle offre à partir des données saisies par l'utilisateur ;
- **update** : met à jour l'offre existante à partir des données saisies par l'utilisateur ;
- **processForm** : appelée par `create` et `update`, cette méthode se charge de traiter le formulaire (validation, repeuplement du formulaire et sérialisation vers la base de données).

Tous les formulaires décrivent le même cycle de vie comme l'illustre le schéma suivant.

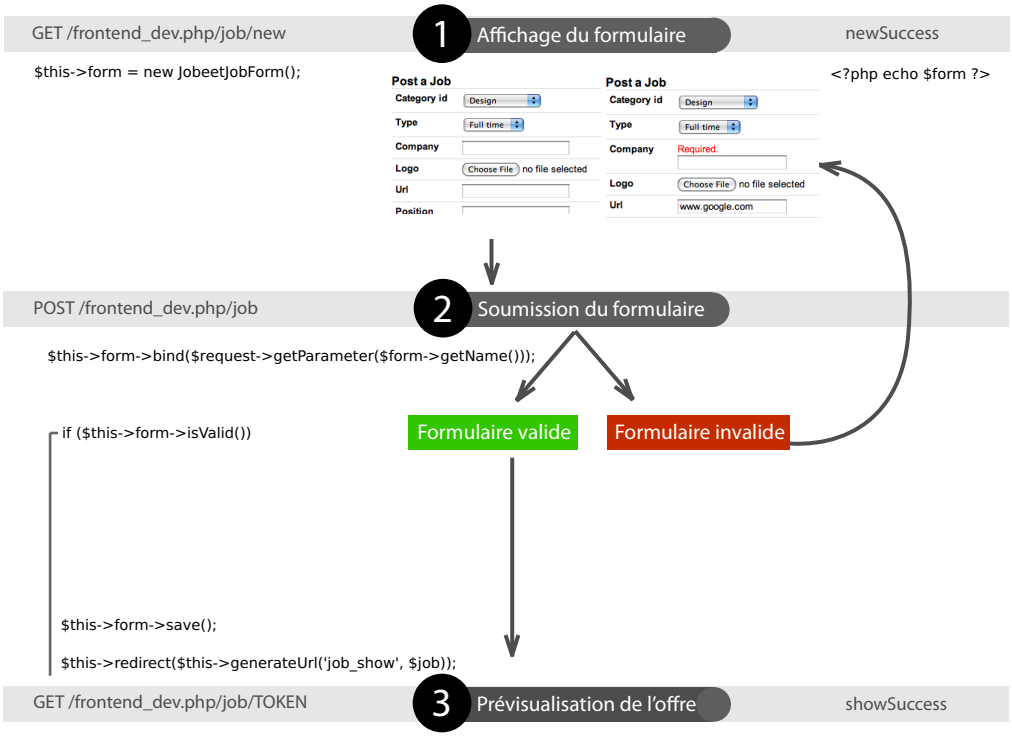


Figure 10–1
Cycle de vie des formulaires dans Symfony

Traiter les formulaires dans les actions

Les sections suivantes s'intéressent à la mise en pratique du formulaire des offres dans les actions. Elles présentent comment le framework Symfony simplifie la manipulation du formulaire tout au long de son cycle de vie en quelques lignes de code élémentaires. La première étape consiste à réduire la méthode de traitement du formulaire `processForm()`.

Simplifier le traitement du formulaire dans le module job

Au chapitre 5, une route Doctrine dédiée au module `job` a été créée. De ce fait, le code de gestion du formulaire peut être simplifié à celui ci-dessous :

Contenu du fichier 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();

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

Comprendre le cycle de vie du formulaire

Quelques explications s'imposent pour comprendre tout le sens de ce code. Que se passe-t-il lorsque l'utilisateur crée ou édite une offre ?

Lorsque celui-ci navigue sur la page /job/new, une nouvelle instance du formulaire est créée et passée au template (action `new`). Puis, dès que l'utilisateur soumet le formulaire (action `create`), ce dernier est initialisé avec les valeurs qu'il a saisies et le processus de validation est mis en route.

À partir du moment où le formulaire est initialisé, il est possible de contrôler sa validité à l'aide de la méthode `isValid()`. Si le formulaire est valide (`isValid()` retourne `true`), alors la nouvelle offre est enregistrée en base de données (`$form->save()`) et l'utilisateur est automatiquement redirigé vers la page de prévisualisation. Dans le cas contraire, le template `newSuccess.php` est réaffiché avec les valeurs saisies ainsi que les messages d'erreur générés.

La modification d'une offre existante est sensiblement la même. La seule différence entre les actions `new` et `edit` réside dans le fait que l'objet `JobeetJob` à modifier est passé comme premier argument du constructeur du formulaire. Il servira alors à définir les valeurs par défaut de chaque widget dans le template. L'ensemble de ces valeurs correspond à un objet pour les formulaires Doctrine, alors qu'il s'agit d'un simple tableau pour les formulaires traditionnels.

Astuce Changer le template par défaut d'une action

La méthode `setTemplate()` change le template utilisé par défaut pour une action donnée. Si le formulaire soumis n'est pas valide, les méthodes `create` et `update` utilisent le même template dans la mesure où les actions `new` et `edit` réaffichent le formulaire avec ses messages d'erreur.

Définir les valeurs par défaut d'un formulaire généré par Doctrine

Il existe deux manières de définir les valeurs par défaut d'un formulaire de création. La première consiste à déclarer les valeurs dans le schéma de définition de la base de données, tandis que la seconde invite à passer un objet `JobeetJob` prémodifié au constructeur du formulaire.

Le code ci-dessous initialise la valeur par défaut (`full-time`) du widget `type` du formulaire à l'aide d'un objet prérempli.

Redéfinition de la méthode executeNew() du fichier 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);
}
```

Quand le formulaire est initialisé avec les valeurs postées, les valeurs par défaut sont automatiquement remplacées par les données saisies. En effet, en cas d'erreur de validation, ces dernières servent à repeupler le formulaire.

Protéger le formulaire des offres par l'implémentation d'un jeton

À présent, tout doit fonctionner correctement mais il reste encore un dernier point à régler : le jeton correspondant ici au champ `token`. Pour le moment, ce dernier doit être rempli manuellement par l'utilisateur. Bien évidemment, le principe du jeton, c'est d'être généré automatiquement lorsque la nouvelle offre est créée. Ce n'est donc plus à l'utilisateur de fournir lui-même cette donnée.

Générer le jeton automatiquement à la création

Le moyen le plus simple et le plus sûr pour y parvenir est de réaliser la génération du jeton dans la méthode `save()` de la classe `JobeetJob`. La valeur du jeton doit être définie juste avant que l'objet soit sérialisé en base de données.

Surcharge de la méthode save() du fichier // lib/model/doctrine/JobeetJob.class.php

```
public function save(Doctrine_Connection $con = null)
{
  // ...

  if (!$this->getToken())
  {
    $this->setToken(sha1($this->getEmail().rand(11111, 99999)));
  }

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

Désormais, le champ `token` peut être retiré de la configuration du formulaire en toute sécurité.

Suppression du champ token dans le fichier 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']
    );

    // ...
  }

  // ...
}
```

Redéfinir la route d'édition de l'offre grâce au jeton

Si l'on se souvient des cas d'utilisation du chapitre 2, une offre d'emploi est éditable à condition que l'utilisateur connaisse le jeton associé. Pour le moment, il est très facile d'éditer ou de supprimer n'importe quelle offre en devinant seulement son URL. En effet, l'URL qui mène au formulaire d'édition repose sur le schéma `job/ID/edit` où la valeur de `ID` correspond à la clé primaire de l'offre dans la base de données.

Par défaut, une route `sfDoctrineRouteCollection` compose les URLs à partir de la clé primaire, mais il est bien sûr possible de remplacer cette dernière par n'importe quelle colonne unique en passant l'option `column`.

Définition de la route job dans le fichier apps/frontend/config/routing.yml

```
job:
  class:        sfDoctrineRouteCollection
  options:      { model: JobeetJob, column: token }
  requirements: { token: \w+ }
```

Il faut remarquer au passage que la contrainte du paramètre `token` a été modifiée également pour correspondre à n'importe quelle chaîne de caractères. En effet, le format obligatoire par défaut de l'option `column` doit être un entier positif en guise de clé unique.

Désormais, toutes les routes relatives aux offres d'emploi, mis à part la route `job_show_user`, intègrent le jeton. Par exemple, la route qui mène à l'édition d'une offre est formatée de la façon suivante :

```
http://jobeet.localhost/job/TOKEN/edit
```

Enfin, il ne reste plus qu'à modifier le lien « Edit » du template `showSuccess.php`.

Construire la page de prévisualisation

La page de prévisualisation est exactement la même que la page de consultation d'une offre. Grâce au routage, si l'utilisateur arrive avec le bon jeton, ce dernier sera accessible dans le paramètre `token` de la requête. D'autre part, si l'utilisateur entre avec une URL contenant le bon jeton, une barre d'administration sera ajoutée au-dessus de l'offre. Il suffit alors tout simplement de modifier le début du template `showSuccess.php` afin que celui-ci puisse accueillir la barre d'administration. Après cela, le lien `edit` situé en pied de page n'a plus qu'à être retiré.

Code à ajouter au début du fichier 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; ?>
```

L'étape suivante consiste à créer le template partiel `_admin.php` en lui ajoutant le contenu suivant.

```
<!-- 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 à étudier. Néanmoins la plupart de celui-ci est très facile à comprendre. Afin de rendre le template plus lisible et compréhensible, un certain nombre de méthodes raccourcies a été ajouté à la classe `JobeetJob`.

Nouvelles méthodes de la classe lib/model/doctrine/JobeetJob.class.php

```
public function getTypeName()
{
  $types = Doctrine::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 floor((strtotime($this->getExpiresAt()) - time()) /
86400);
}
```

La barre d'administration affiche les différentes actions en fonction du statut de l'offre d'emploi.

Figure 10–2
État de la barre d'administration d'une offre non active

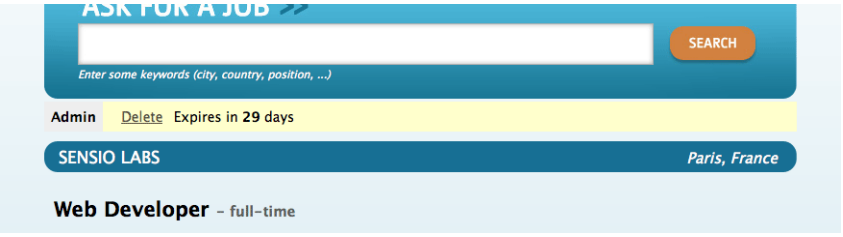


Figure 10–3
État de la barre d'administration d'une offre active

La barre d'activation du second écran sera mise en place dès la prochaine section.

Activer et publier une offre

Préparer la route vers l'action de publication

Dans la section précédente se trouve un lien pour publier une offre d'emploi. Celui-ci a besoin d'être modifié pour pointer vers l'action `publish`. Au lieu de créer une nouvelle route, il suffit de configurer la route existante `job` comme le montre le code ci-dessous.

Configuration de la route job dans le fichier apps/frontend/config/routing.yml

```
job:
  class:   sfDoctrineRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: put }
  requirements:
    token: \w+
```

L'option `object_actions` prend un tableau des actions additionnelles pour l'objet donné. Ainsi, le lien de publication d'une offre peut désormais être modifié.

Extrait du contenu du fichier apps/frontend/modules/job/templates/_admin.php

```
<li>
  <?php echo link_to('Publish', 'job_publish', $job,
array('method' => 'put')) ?>
</li>
```

Implémenter la méthode executePublish()

La dernière étape consiste à créer l'action `executePublish()` dans le fichier `actions.class.php` du module job.

Méthode executePublish() à ajouter au fichier 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')));

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

Le lecteur assidu aura remarqué que le lien « Publish » est soumis à l'aide de la méthode HTTP `PUT`. Pour simuler cette méthode `PUT`, le lien est automatiquement converti en un formulaire quand on clique dessus grâce au JavaScript.

D'autre part, comme la protection CSRF (Cross-Site Request Forgeries) a été activée pour Jobeet lors du premier chapitre, le helper `link_to()` intègre un jeton CSRF au lien, tandis que la méthode `checkCSRFProtection()` de l'objet requête s'assure de sa validité au moment de son envoi.

BONNE PRATIQUE

Précisions sur la faille de sécurité CSRF

Il s'agit d'une faille de sécurité très répandue mais malheureusement très rarement prise en compte dans le développement d'applications. Cette faille profite de la confiance qu'a l'utilisateur dans les systèmes d'authentification pour effectuer certaines actions à son insu. L'un des moyens les plus efficaces pour se protéger de ces attaquesreste l'implémentation d'un jeton, comme le fait le helper `link_to()` par défaut.

Implémenter la méthode publish() de l'objet JobeetJob

La méthode `executePublish()` du contrôleur utilise une nouvelle méthode `publish()` appliquée sur l'objet `JobeetJob`. Celle-ci se résume simplement au code ci-dessous.

Déclaration de la méthode publish() dans le fichier lib/model/doctrine/JobeetJob.class.php

```
public function publish()
{
  $this->setIsActivated(true);
  $this->save();
}
```

La nouvelle fonctionnalité de publication d'offre d'emploi peut maintenant être testée dans un navigateur. Néanmoins, il reste encore quelque chose à fixer : Les offres non activées ne doivent pas être accessibles, ce qui signifie qu'elles ne doivent plus être affichées sur la page d'accueil de Jobeet, et ne doivent pas non plus être atteignables par leur URL.

Empêcher la publication et l'accès aux offres non actives

Dans les précédents chapitres, une méthode `addActiveJobsQuery()` avait été créée pour restreindre un critère aux seules offres actives. Il suffit alors d'éditer et d'ajouter de nouvelles contraintes à la fin de la méthode, comme le montre le code suivant.

Méthode addActiveJobsQuery() du fichier lib/model/doctrine/JobeetJobTable.class.php

```
public function addActiveJobsQuery(Doctrine_Query $q = null)
{
  // ...

  $q->andWhere($alias . '.is_activated = ?', 1);

  return $q;
}
```

C'est fini ! Il ne reste plus qu'à tester dans le navigateur que l'ensemble se comporte correctement. Désormais, toutes les offres non actives ont disparu de la page d'accueil et ne sont plus accessibles en devinant leur URL. Elles restent en revanche atteignables si quelqu'un connaît le jeton de l'offre et s'en sert dans l'URL. Dans ce cas précis, ce sera la page de prévisualisation de l'offre et sa barre d'administration qui seront affichées à l'écran.

C'est l'un des principaux avantages du motif de conception MVC et de la refactorisation qui a été réalisée tout au long de ce parcours. Seulement un unique changement dans une seule méthode a permis d'appliquer la nouvelle contrainte sur l'ensemble de l'application.

Lorsque la méthode `getWithJobs()` a été créée, l'utilisation de la méthode `addActiveJobsQuery()` a été oubliée, ce qui signifie qu'il faille l'éditer et ajouter la nouvelle contrainte.

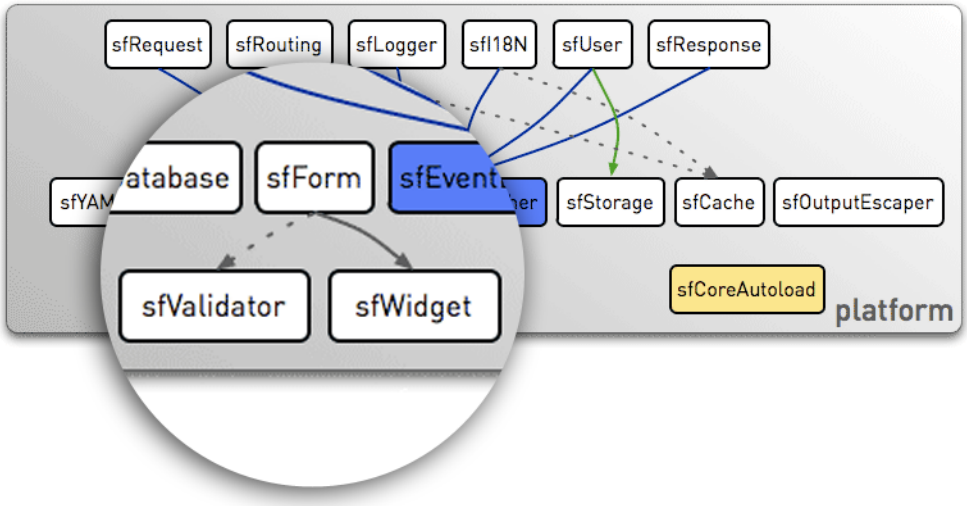
```
class JobeetCategoryTable extends Doctrine_Table
{
  public function getWithJobs()
  {
    // ...

    $q->andWhere('j.is_activated = ?', 1);

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

En résumé…

Ce chapitre a abordé toute une série de nouvelles informations, qui vous ont permis d'acquérir une meilleure connaissance du framework de formulaires de Symfony.

Un point fondamental reste encore non traité au terme de ces quelques pages puisqu'en effet, aucun nouveau test n'a été implémenté pour les nouvelles fonctionnalités. Dans la mesure où l'écriture de tests est un enjeu crucial dans le développement d'une application, ce sera le tout premier thème abordé au chapitre suivant.

chapitre 11


sfRequest
sfRouting
sfLogger
sfI18N
sfUser
sfResponse
atabase
sfForm
sfEvent
sfYAM
sfStorage
sfCache
sfOutputEscaper
sfValidator
sfWidget
sfCoreAutoload
platform

Tester les formulaires

Les formulaires sont des composants complexes à gérer dans une application web. Heureusement, le chapitre précédent a montré comment Symfony en facilite grandement la gestion à l'aide de son framework interne.

Ce chapitre approfondit encore les connaissances liées aux formulaires en montrant les manières de les tester fonctionnellement.

MOTS-CLÉS :

- Tests fonctionnels
- Sécurité XSS et CSRF
- Créer des tâches automatiques

À ce stade, les utilisateurs ont la capacité de naviguer sur l'application à la recherche d'offres d'emploi, et d'en ajouter de nouvelles grâce aux formulaires. Néanmoins, le formulaire de création d'une nouvelle offre n'a pas encore été testé pour s'assurer qu'il se comporte normalement. Ce chapitre couvre avant tout les notions de tests fonctionnels des formulaires, et apporte quelques astuces supplémentaires à propos du framework de formulaire.

Utiliser le framework de formulaires de manière autonome

Les composants de Symfony sont relativement découplés, ce qui signifie que la plupart d'entre eux sont exploitables de manière autonome à l'extérieur du framework. Le framework de formulaire en fait partie puisqu'il ne dispose d'aucune dépendance avec le reste de Symfony. Les classes de formulaire, les widgets ainsi que les validateurs sont ainsi réutilisables hors de l'environnement Symfony. Pour ce faire, il suffit de récupérer les répertoires `lib/form/`, `lib/widget/` et `lib/validator/` qui se trouvent dans le répertoire `lib/vendor/symfony/` d'un projet.

Parmi ces composants autonomes figure également le framework interne de routage, accessible dans le répertoire `lib/routing/`. Celui-ci permet de profiter pleinement et gratuitement des URLs bien formées.

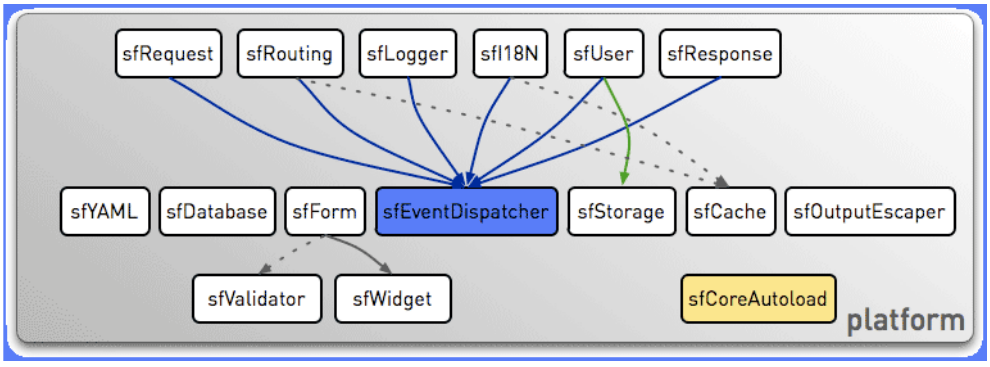


Figure 11–1
Diagramme des composants indépendants de Symfony

Après cette brève introduction concernant la facilité d'utilisation du framework de formulaire en dehors de l'environnement Symfony, il est temps d'entrer dans le vif du sujet. Il s'agit de poursuivre le développement de l'application en s'intéressant aux tests fonctionnels des formulaires. Le neuvième chapitre a présenté globalement le but et le principe de fonctionnement des scénarios de tests fonctionnels. Jusqu'à mainte-

nant, ces derniers ont essentiellement servi à analyser le contenu HTML de la réponse grâce notamment aux sélecteurs CSS 3. Cet outil recèle encore bien d'autres fonctionnalités puisqu'il offre également la capacité de tester n'importe quelle classe de formulaire.

Écrire des tests fonctionnels pour les classes de formulaire

Les prochaines sections présentent les différents outils du framework de tests fonctionnels qui permettent de tester les formulaires. Ces derniers assurent au développeur que le formulaire qu'il a écrit se comporte convenablement en simulant le remplissage des champs ainsi que les téléchargements de fichiers, et en diagnostiquant les erreurs générées par les validateurs.

Tester l'envoi du formulaire de création d'offre

La première étape du processus de test du formulaire de création d'une nouvelle offre consiste à s'assurer que ce dernier est bien envoyé avec toutes les valeurs obligatoires renseignées. Pour commencer, le code ci-dessous doit être ajouté à la fin du fichier de tests fonctionnels `jobActionsTest.php`.

Code à ajouter à la fin du fichier 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()
;
```

Au neuvième chapitre, la méthode `click()` de l'objet `sfTestFunctional` a été utilisée pour simuler les clics sur des liens présents dans la page. Cette méthode `click()` fonctionne de la même manière pour soumettre un formulaire. En effet, elle accepte en second argument un tableau associatif des valeurs à envoyer pour chaque champ. Tel un véritable navigateur, l'objet `$browser` fusionnera les valeurs par défaut du formulaire avec celles qui ont été soumises.

Renommer le nom des champs du formulaire

Cependant, il faut connaître à l'avance le nom des champs pour lesquels les valeurs doivent être transmises. En ouvrant le code source HTML ou bien en utilisant la fonctionnalité *Formulaires > Afficher les détails du formulaire* de la Firefox Web Developer Toolbar sur le formulaire en question, on remarque que le nom du champ `company` est en fait `jobeet_job[company]`.

Lorsque PHP rencontre un champ de saisie avec un nom comme `jobeet_job[company]`, il le convertit automatiquement en un tableau dont le nom est `jobeet_job`. Ce format de nommage des champs est un peu complexe et ne facilite pas forcément la tâche. En ajoutant la ligne de code suivante à la fin de la méthode `configure()` de la classe `JobeetJobForm`, les noms des champs suivront alors le format `$job[%s]`.

```
$this->widgetSchema->setNameFormat('job[%s]');
```

Soumettre le formulaire à l'aide de la méthode click()

Après ce changement, le nom du champ `company` devrait être `job[company]`. Il est maintenant l'heure de tester le clic sur le bouton *Preview your job* en fournissant un tableau de valeurs valides au formulaire, par le biais du second argument de la méthode `click()`.

Code de test de l'envoi du formulaire à ajouter au fichier 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 code ci-dessus ne réalise rien d'extraordinaire. En effet, il se charge simplement de transmettre les données du formulaire lorsque l'on clique sur le bouton *Preview your job*, puis de vérifier que la page suivante correspond bien à l'action `create` du module `job`. La simulation des envois de fichiers est elle aussi prise en compte en fournissant le chemin absolu vers un fichier comme le montre le champ `logo`.

Découvrir le testeur sfTesterForm

L'objet `sfTesterForm` est un testeur qui permet de vérifier les données d'un formulaire comme le réalise le testeur `sfTesterResponse` avec la réponse générée.

Tester si le formulaire est erroné

Le testeur `sfTesterForm` fournit une méthode bien pratique, `hasErrors()`, pour tester si oui ou non le formulaire a généré des erreurs d'après les données qui lui ont été transmises. L'exemple de code ci-dessous illustre son utilisation.

```
with('form')->begin()->
  hasErrors(false)->
end();
```

Les méthodes de l'objet sfTesterForm

L'objet `sfTesterForm` dispose de bien d'autres méthodes utiles pour récupérer des informations sur l'état du formulaire, comme les erreurs générées. Le tableau suivant dresse une liste exhaustive des méthodes de cet objet.

Tableau 11–1 Liste des méthodes de l'objet sfTesterForm

Répertoire	Description
`getForm()`	Retourne l'objet de formulaire courant
`hasErrors()`	Retourne si oui ou non le formulaire a des erreurs
`hasGlobalError()`	Retourne si oui ou non le formulaire a des erreurs globales
`isError($field)`	Retourne si oui ou non le champ donné a des erreurs
`debug()`	Affiche tout l'état du formulaire en cours

Déboguer un formulaire

Lorsque un test échoue, le premier moyen de faciliter la découverte du bug est d'utiliser la méthode `debug()` sur le testeur `sfTesterResponse` afin d'obtenir la réponse générée par le serveur. Dans le cas d'un formulaire, ce n'est véritablement pas pratique dans la mesure où il faut plonger soi-même dans le code HTML à la recherche des erreurs générées pour chaque champ.

Heureusement, le testeur `sfTesterForm` fournit la méthode `debug()` qui permet d'imprimer en sortie l'état complet du formulaire avec les erreurs levées. Le bout de code ci-dessous montre comment utiliser cette méthode sur le testeur de formulaire.

```
with('form')->debug()
```

Tester les redirections HTTP

Dans Jobeet, lorsque le formulaire soumis par l'utilisateur est valide, ce dernier est automatiquement redirigé vers la page de consultation de l'offre se trouvant à l'action `show`. De ce fait, il est nécessaire de s'assurer que la redirection a bien eu lieu grâce aux méthodes `isRedirected()` et `followRedirect()` comme le montre l'exemple de code ci-dessous.

```
isRedirected()->
followRedirect()->

with('request')->begin()->
  isParameter('module', 'job')->
  isParameter('action', 'show')->
end();
```

La méthode `isRedirected()` indique si oui ou non il y a eu une redirection tandis que la méthode `followRedirect()` suit cette dernière. Pourquoi la classe du navigateur ne suit-elle pas automatiquement la redirection ? La raison est simple : c'est pour laisser au développeur la liberté d'analyser l'état des objets avant la redirection.

Tester les objets générés par Doctrine

Dans la plupart des cas, les formulaires sont destinés à récupérer les informations de l'utilisateur afin de les stocker dans la base de données. Dans Symfony, c'est exactement ce que font les formulaires Doctrine puisqu'ils permettent de sauvegarder automatiquement les informations saisies dans une table. Cependant, comment s'assurer que les informations ont bien été enregistrées et que les valeurs de chaque champ correspondent à ce que l'on souhaite ?

Activer le testeur sfTesterDoctrine

Le framework interne de tests fonctionnels de Symfony introduit un nouveau testeur, l'objet `sfTesterDoctrine`, qui permet d'interroger une base de données. Contrairement aux testeurs vus jusqu'à présent, le testeur Doctrine n'est pas référencé par défaut dans l'objet `sfTestFunctional`. Pour l'activer, il suffit de le définir explicitement comme l'explique le code ci-dessous.

```
$browser->setTester('doctrine', 'sfTesterDoctrine');
```

Tester l'existence d'un objet Doctrine dans la base de données

Dans le cadre de l'application Jobeet, il est intéressant de contrôler que l'offre d'emploi a bien été créée et que la colonne `is_activated` contient la valeur `false` dans la mesure où l'utilisateur ne l'a pas encore publiée.

Grâce au testeur `sfTesterDoctrine` et à sa méthode `check()`, il est tout à fait possible de vérifier l'existence d'un ou de plusieurs objets dans la base de données qui correspondent à un critère donné. Le code suivant illustre l'utilisation du testeur Doctrine pour contrôler la création de la nouvelle offre d'emploi.

```
with('doctrine')->begin()->
  check('JobeetJob', array(
    'location'     => 'Atlanta, USA',
    'is_activated' => false,
    'is_public'    => false,
  ))->
end()
```

Le critère de recherche de la méthode `check()` peut s'exprimer de deux manières différentes. Il s'agit en effet de passer soit un tableau associatif comme dans l'exemple ci-dessus, soit une instance de la classe `Doctrine_Query` dans le cas de requêtes plus complexes. De plus, un troisième argument facultatif peut lui être transmis. Si ce dernier est booléen, il spécifie s'il faut tester l'existence (`true` par défaut) ou bien l'absence (`false`) de l'objet dans la base de données. En revanche, si un nombre entier est passé en paramètre, la méthode `check()` vérifiera que le critère correspond au nombre de résultats.

Tester les erreurs des champs du formulaire

Tester un formulaire avec des valeurs valides ne suffit pas. Il faut aussi s'assurer qu'il se comporte correctement lorsque des données invalides, voire potentiellement dangereuses, lui sont transmises. La première vérification consiste donc à s'assurer que ces données ne sont pas considérées

comme valides par les validateurs qu'elles traversent, et que ces derniers génèrent effectivement les messages d'erreur adéquats.

La méthode isError() pour le contrôle des champs

La méthode `isError()` du testeur `sfTesterForm` permet de tester si un champ du formulaire est erroné en lui passant en paramètres le nom du champ concerné et le code d'erreur du validateur (`required`, `invalid`, `max_length`, `min_length`...).

Le code qui suit vérifie que le formulaire lève trois erreurs pour les champs `description`, `how_to_apply` et `email`. Pour les deux premiers, il s'agit de s'assurer que leur valeur respective ne peut rester vide tandis que pour le champ `email`, on vérifie que la valeur soumise ne correspond pas un format d'adresse e-mail valide.

```
$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()
;
```

En prenant une valeur entière plutôt qu'un booléen comme argument, la méthode `hasErrors()` vérifie que le formulaire a généré exactement ce nombre d'erreurs. Dans le cas du test d'un formulaire avec des valeurs invalides, il est inutile de tester chaque champ. Seuls les plus sensibles ou les plus spécifiques peuvent être contrôlés pour valider que le formulaire est bien mis en échec avec des données invalides. De plus, les composants du framework interne de formulaire ont déjà été testés, ce qui donne l'assurance qu'ils se comportent correctement.

Les véritables messages d'erreur peuvent également être testés en utilisant la méthode `checkElement()` du testeur `sfTesterResponse`. Cette technique est particulièrement recommandée lorsque les formulaires ont un layout personnalisé. Dans l'application Jobeet, le layout du formulaire est celui par défaut, c'est pourquoi les messages d'erreur ne sont pas testés.

Tester la barre d'administration d'une offre

L'étape suivante consiste à tester les liens de la barre d'administration de la page de prévisualisation d'une offre. Lorsqu'une offre n'est pas encore activée, l'utilisateur a toujours la capacité de l'éditer, de la publier ou bien de la supprimer en cliquant sur le lien correspondant. Afin de tester chacun de ces liens, il est nécessaire de créer une nouvelle offre. Bien entendu, il est hors de question de copier/coller le code de création d'une offre pour chacun des cas. Ce serait en effet laborieux, difficilement maintenable et une véritable perte de temps. L'idéal est donc de mutualiser ce code dans une nouvelle méthode de la classe `JobeetTestFunctional` comme le montre le code suivant.

Méthode createJob() à ajouter au fichier 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 nouvelle méthode `createJob()` crée une nouvelle offre, suit la redirection et retourne le navigateur afin de ne pas casser l'interface fluide. Un tableau de valeurs peut également lui être fourni en paramètre. Celui-ci sera fusionné avec les valeurs par défaut afin de pouvoir redéfinir certaines d'entre elles facilement.

Forcer la méthode HTTP d'un lien

Forcer l'utilisation de la méthode HTTP PUT

Au chapitre précédent, le lien *Publish* a été configuré pour fonctionner avec la méthode HTTP `PUT`. Comme les navigateurs ne supportent pas les requêtes `PUT`, le helper `link_to()` convertit le lien en un formulaire à l'aide d'un script JavaScript.

Cependant, le navigateur de tests fonctionnels est incapable d'exécuter le moindre code JavaScript, c'est pourquoi la méthode `PUT` doit être forcée manuellement en la passant comme troisième argument facultatif de la méthode `click()`.

De plus, le helper `link_to()` embarque un jeton CSRF puisque la protection CSRF a été activée lors du premier chapitre. L'option `_with_csrf` simule ce jeton.

```
$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()
;
```

Il est alors possible de reproduire cet exemple pour tester le lien *Delete* qui nécessite, quant à lui, le recours à la méthode HTTP `DELETE`.

Forcer l'utilisation de la méthode HTTP DELETE

Le fonctionnement du lien *Delete* est exactement le même que celui du bouton *Publish*. Il s'agit en effet de fournir cette fois-ci la valeur `delete` au paramètre `method`, puis de s'assurer que l'objet a bien été supprimé de la base de données grâce au testeur `sfTesterDoctrine`.

```
$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()
;
```

La section suivante explique en quoi l'écriture de tests fonctionnels est un excellent moyen de découvrir des bogues ou bien de détecter des fonctionnalités encore non implémentées…

Écrire des tests fonctionnels afin de découvrir des bogues

Lorsqu'une offre est publiée, il devient impossible de l'éditer par la suite. Bien que le lien *Edit* ait totalement disparu de la page de prévisualisation, il est toujours possible d'avoir accès au formulaire d'édition de l'offre via l'URL. Le seul moyen efficace de le vérifier est bien sûr d'écrire quelques tests automatiques.

Simuler l'autopublication d'une offre

La première étape consiste à éditer la méthode `createJob()` en lui ajoutant un nouvel argument facultatif permettant de forcer l'autopublication de l'offre avant d'écrire une nouvelle méthode `getJobByPosition()`. Cette dernière se charge de récupérer et de retourner une offre d'emploi à partir de la valeur de la colonne `position`.

Méthodes createJob() et getJobByPosition() du fichier 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();
  }

  // ...
}
```

Contrôler la redirection vers une page d'erreur 404

Lorsqu'une offre est publiée, l'accès au formulaire d'édition ne doit plus être possible et doit conduire vers une page d'erreur 404. Pour s'en assurer, il est nécessaire d'écrire quelques tests fonctionnels comme le présente le morceau de code ci-dessous.

```
$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()
;
```

Les trois lignes de code en exergue décrivent le scénario de test suivant :

1. une nouvelle offre est créée et autopubliée ;
2. le jeton de celle-ci est récupéré pour construire l'URL du formulaire d'édition et s'y rendre ;

3 le testeur de réponse vérifie que le statut HTTP de la réponse est bel et bien égal à 404 (page introuvable).

En exécutant la suite de tests fonctionnels, le test du statut HTTP de la réponse échoue. Ce n'est finalement pas si étonnant puisque l'implémentation de cette fonctionnalité a été oubliée lors du chapitre précédent.

En somme, les tests automatiques sont un excellent moyen de faire apparaître des effets de bord, des bogues ou bien encore des fonctionnalités importantes non implémentées. L'écriture de tests demande aux développeurs de penser à tous les cas de figure possibles, ce qui n'est pas systématique lorsqu'ils développent chaque fonctionnalité.

Empêcher l'accès au formulaire d'édition lorsque l'offre est publiée

Fixer le bogue découvert dans la section précédente est à présent très simple dans la mesure où le test fonctionnel en donne la solution : rediriger vers une page d'erreur 404 lorsque l'offre demandée est déjà publiée. Pour ce faire, il suffit d'avoir recours à la méthode `forward404If()` dans l'action `executeEdit()` du module `job`.

Méthode executeEdit() du fichier 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 ne garantit pas que tout fonctionne toujours correctement. Un moyen simple de s'en assurer est d'ouvrir le navigateur et de tester toutes les combinaisons possibles pour accéder au formulaire d'édition. C'est une méthode somme toute fastidieuse…

La meilleure façon de procéder consiste bien évidemment à relancer toute la suite de tests fonctionnels car ces derniers permettent d'une part de vérifier que le correctif répond cette fois-ci au test écrit, et d'autre part qu'aucune régression fonctionnelle n'a aussi été entraînée suite à la modification de l'action `executeEdit()`.

BONNE PRATIQUE

Éviter les tâches fastidieuses

Les tâches fastidieuses sont nombreuses dans le développement d'une application web. Néanmoins, il faut se rappeler que si une tâche est fastidieuse et rébarbative, il existe probablement un outil permettant de les réaliser à la place du développeur. C'est un état d'esprit à garder constamment, surtout lors de l'utilisation d'outils aussi complets que Symfony.

Tester la prolongation d'une offre

Dans Jobeet, lorsqu'une offre d'emploi expire dans les cinq prochains jours, ou lorsqu'elle est déjà arrivée à expiration, l'utilisateur a la possibilité de prolonger sa durée de vie de trente jours supplémentaires à compter de la date courante.

Comprendre le problème des offres expirées à réactiver

Tester cette contrainte dans un navigateur n'est pas si facile dans la mesure où la date d'expiration est automatiquement définie à trente jours lors de la création de l'offre. Ainsi, lorsque l'on accède à la page de consultation de l'offre, le lien pour la prolonger n'est pas présent. Bien sûr, il est possible de « pirater » la date d'expiration dans la base de données, ou bien de personnaliser le template pour qu'il affiche toujours le lien, mais c'est à la fois fastidieux et annonciateur d'erreur. On l'aura deviné, écrire des tests est une fois de plus une aide particulièrement appréciée.

Une route dédiée pour prolonger la durée d'une offre

Pour permettre à l'auteur de prolonger son annonce sur le site, la première étape consiste comme d'habitude à penser puis définir la route correspondante. C'est exactement ce que réalise le code ci-dessous en ajoutant une nouvelle route à la collection de routes Doctrine de l'objet.

Ajout de la nouvelle route extend à la collection de routes d'une offre dans le fichier apps/frontend/config/routing.yml

```
job:
  class:   sfDoctrineRouteCollection
  options:
    model:          JobeetJob
    column:         token
    object_actions: { publish: PUT, extend: PUT }
  requirements:
    token: \w+
```

Après cela, il suffit de mettre à jour le template partiel `_admin.php` afin de lui spécifier la méthode `PUT` pour le lien *Extend*.

Extrait du fichier 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; ?>
```

Implémenter la méthode executeExtend() aux actions du module job

Maintenant que la route est proprement définie, le prochain objectif est d'implémenter la nouvelle action `executeExtend()` qui a pour rôle de prolonger la durée de vie d'une offre, à condition que celle-ci arrive bientôt à expiration.

Méthode executeExtend() à ajouter au fichier apps/frontend/modules/job/actions/actions.class.php

```
public function executeExtend(sfWebRequest $request)
{
  $request->checkCSRFProtection();

  $job = $this->getRoute()->getObject();
  $this->forward404Unless($job->extend());

  $this->getUser()->setFlash('notice', sprintf('Your job
validity has been extend until %s.', date('m/d/Y',
strtotime($job->getExpiresAt()))));

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

Le code de cette nouvelle action est très simple à comprendre.

1. La méthode `checkCSRFProtection()` contrôle la validité du jeton transmis suite au clic sur le lien *Extend*.
2. Puis l'offre d'emploi est retrouvée à partir de sa route et de son propre jeton. La méthode `extend()` de l'objet `JobeetJob` prolonge la durée de vie de l'offre pour une nouvelle période de 30 jours à condition que celle-ci arrive à expiration prochainement. Dans le cas contraire, l'utilisateur est automatiquement redirigé vers une page d'erreur 404.
3. Enfin, si tout s'est bien déroulé, l'utilisateur est automatiquement redirigé vers son annonce en profitant d'un message lui informant de la nouvelle date d'expiration de son offre.

Implémenter la méthode extend() dans JobeetJob

À présent, il ne reste plus qu'à implémenter la méthode `extend()` à la classe `JobeetJob` avant de s'assurer que cette nouvelle fonctionnalité est entièrement opérationnelle grâce à quelques scénarios de tests fonctionnels. Le code suivant présente le détail de la méthode `extend()`.

Méthode extend() à ajouter au fichier 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;
  }

  // ...
}
```

Là encore, le code de la méthode est suffisamment clair, intuitif et explicite pour ne pas être commenté davantage. Cette méthode retourne immédiatement `false` si l'offre n'est pas sur le point d'expirer. Dans le cas contraire, la nouvelle date d'expiration est recalculée et l'objet est sérialisé en base de données avant de retourner `true`.

Tester la prolongation de la durée de vie d'une offre

La dernière étape consiste à vérifier avec quelques scénarios de test que la nouvelle fonctionnalité a bien été implémentée et qu'elle ne provoque pas d'effet de bord ou de régression pour les autres fonctionnalités déjà implémentées.

Une offre ne peut être prolongée si elle n'expire pas bientôt

Le premier test à effectuer doit vérifier que l'utilisateur ne peut prolonger la durée de vie de son annonce si cette dernière n'est pas sur le point d'expirer. Le test contrôle que l'utilisateur est redirigé automatiquement vers une page d'erreur 404 comme le montre le bout de code ci-dessous.

```
$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()
;
```

On note ici l'utilisation de la méthode `call()` à la place de `get()` pour exécuter la prolongation de la durée de vie de l'offre. `call()` sert effectivement à appeler des URLs dont la méthode HTTP est différente de `GET` ou `POST`.

Une offre peut être prolongée uniquement si elle expire bientôt

Le second test consiste à vérifier cette fois-ci qu'une offre prête à expirer peut être prolongée. C'est le scénario que teste le code suivant.

```
$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();
$browser->test()->is(
  date('y/m/d', strtotime($job->getExpiresAt())),
  date('y/m/d', time() + 86400 *
sfConfig::get('app_active_days'))
);
```

Quelques nouvelles explications s'imposent ici. De la même manière qu'avec le scénario précédent, la méthode `call()` appelle l'URL avec la méthode PUT et un jeton. On constate que l'utilisateur est bien redirigé mais la redirection n'est pas suivie afin d'effectuer quelques tests supplémentaires.

Tout d'abord, l'offre d'emploi est rafraîchie (`$job->refresh()`) dans le but de prendre en compte les modifications. Ensuite, le timestamp sauvegardé en base de données est comparé avec celui de la date courante plus trente jours à l'aide de l'objet `lime_test` du navigateur.

Sécuriser les formulaires

Le framework interne de formulaire de Symfony intègre nativement des points de sécurité à différents endroits. La création de jeton unique pour chaque formulaire, l'échappement automatique des données contre les failles de type « Cross Site Scripting » (XSS) ou bien encore l'injection de transactions SQL lors de la sauvegarde d'un objet en base de données sont d'autant de points sensibles à protéger. Heureusement, Symfony soulage le développeur de cette tâche.

Sérialisation d'un formulaire Doctrine

Les formulaires Doctrine sont très simples à utiliser dans la mesure où ils automatisent une grande partie du travail du développeur. Par exemple, sérialiser un formulaire Doctrine en base de données consiste en un simple appel à la méthode `save()` du formulaire : `$form->save()`.

Mais concrètement, que se passe-il dans cette méthode ? La méthode `save()` déroule en fait les étapes suivantes une à une :

1. démarrage d'une transaction SQL car les formulaires imbriqués Doctrine sont tous sauvegardés d'un seul coup ;
2. traitement des valeurs soumises en appelant les méthodes `updateCOLUMNColumn()` si elles existent ;
3. appel de la méthode `fromArray()` de l'objet Doctrine pour mettre à jour les valeurs des colonnes ;
4. sauvegarde de l'objet en base de données ;
5. validation de la transaction.

L'utilisation de la méthode `fromArray()` peut potentiellement engendrer une faille de sécurité non négligeable pour le système d'information si elle est mal utilisée. Heureusement, le framework interne de formulaire de Symfony réalise les traitements adéquats pour s'en prémunir. La section suivante explique en détail comment cette sécurité est implémentée.

Sécurité native du framework de formulaire

La méthode `fromArray()` prend un tableau de valeurs en paramètre et met à jour les valeurs des colonnes correspondantes. En quoi est-ce que cela représente un éventuel problème de sécurité ? Que se passe-t-il si quelqu'un essaie de soumettre sans l'autorisation nécessaire une valeur à une colonne ? Par exemple, est-il possible de forcer la valeur de la colonne `token` directement dans le formulaire ?

Le meilleur moyen de s'en assurer est d'écrire un nouveau test qui simule la soumission d'un formulaire de création d'offre d'emploi dans lequel se trouve un champ supplémentaire `token`.

Code à ajouter au fichier 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()
;
```

Lorsque le formulaire est soumis, une erreur globale de type `extra_fields` est levée. C'est en effet parce que par défaut, les formulaires n'acceptent pas la présence de champs supplémentaires parmi les valeurs transmises. C'est aussi pour cette raison que tous les champs de formulaire doivent posséder un validateur associé.

Toutefois cette mesure de sécurité peut être outrepassée en fixant l'option `allow_extra_fields` à `true`.

```
class MyForm extends sfForm
{
  public function configure()
  {
    // ...

    $this->validatorSchema->setOption('allow_extra_fields',
true);
  }
}
```

Le test devrait à présent passer bien que la valeur du champ `token` a été filtrée. Ainsi, il n'est toujours pas possible de franchir cette mesure de sécurité. Pour rendre véritablement possible ce cas de figure, il suffit de fixer la valeur de l'option `filter_extra_fields` à `false`.

```
$this->validatorSchema->setOption('filter_extra_fields',
false);
```

Les tests écrits dans cette section ne sont qu'à but démonstratif. Ils peuvent être retirés du projet Jobeet dans la mesure où il n'est pas nécessaire de valider des fonctionnalités de Symfony déjà testées.

ASTUCE Modifier un formulaire avec la Firefox Web Developer Toolbar

Des outils comme la célèbre Firefox Web Developer Toolbar permettent de modifier les valeurs des champs d'un formulaire mais aussi d'en ajouter de nouveau simplement depuis le navigateur.

Se protéger contre les attaques CSRF et XSS

Au cours du premier chapitre, l'application `frontend` a été créée à partir de la ligne de commande suivante.

```
$ php symfony generate:app --escaping-strategy=on --csrf-
secret="Unique$ecret" frontend
```

L'option `--escaping-strategy` active la protection contre les failles de sécurité XSS, ce qui signifie que toutes les variables utilisées dans les templates sont échappées par défaut. Lors de la création d'une nouvelle offre d'emploi, si la description de cette dernière contient des balises HTML, alors la description de l'annonce dans la page de consultation de détail sera rendue par Symfony comme étant du texte plein et non interprétée comme du code HTML par le navigateur.

L'option `--csrf-secret` active la protection contre les failles de sécurité CSRF (prononcer « Sea Surf »). Lorsque cette option est activée, tous les formulaires intègrent un champ caché `_csrf_token`.

La stratégie d'échappement et la clé secrète contre les CSRF peuvent être modifiées à n'importe quel moment en éditant manuellement le fichier de configuration `apps/frontend/config/settings.yml`. De la même manière que le fichier `databases.yml`, les paramètres de configuration sont configurables par environnement.

```
all:
  .settings:
    # Form security secret (CSRF protection)
    csrf_secret: Unique$ecret

    # Output escaping settings
    escaping_strategy: on
    escaping_method:   ESC_SPECIALCHARS
```

Avant de terminer ce chapitre, il est intéressant de passer quelques minutes sur la création de tâches automatiques propres au projet. Dans le cadre de Jobeet, il s'agit d'écrire une tâche qui se charge de supprimer de la base de données toutes les offres d'emploi expirées depuis un certain nombre de jours.

Les tâches automatiques de maintenance

Symfony est un framework web, mais il n'en est pas moins livré avec un outil fonctionnant en ligne de commande. Ce dernier a déjà été utilisé à plusieurs reprises pour créer l'architecture par défaut du projet et de l'appli-

cation, ou bien encore pour générer quelques fichiers du modèle. Ajouter une nouvelle tâche est simplissime dans la mesure où les outils utilisés par la ligne de commande Symfony sont déjà intégrés dans le framework.

Créer la nouvelle tâche de maintenance jobeet:cleanup

Lorsque l'utilisateur crée une nouvelle offre d'emploi, il doit impérativement l'activer pour la mettre en ligne. À défaut et avec le temps, la base de données continuera de grossir avec des annonces hors ligne. Il devient donc utile de créer une tâche automatique qui se charge de supprimer toutes les offres non publiées de la base de données. Pour éviter d'avoir à la lancer régulièrement à la main, cette tâche doit être exécutée périodiquement sous forme d'une tâche automatique planifiée (*cron job*).

Nouvelle tâche automatique de maintenance à créer dans le fichier 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);

    $nb = Doctrine::getTable('JobeetJob')
                    ->cleanup($options['days']);
```

```
    $this->logSection('doctrine',
                      sprintf('Removed %d stale jobs', $nb));
  }
}
```

La configuration de la tâche est réalisée dans la méthode `configure()`. Chaque nouvelle tâche doit avoir un nom unique (`namespace:name`), et peut prendre des arguments et des options en guise de paramètres. Pour en savoir plus sur les tâches internes de Symfony, il suffit de regarder dans le répertoire `lib/task` du framework Symfony pour analyser le code source de quelques unes d'entre elles.

Comme la nouvelle classe vient tout juste d'être créée, il faut bien évidemment réactualiser le cache de Symfony pour qu'elle soit prise en compte.

```
$ php symfony cc
```

La tâche `jobeet:cleanup` définit deux options : `--env` et `--days` avec pour chacune une valeur par défaut. Tout le manuel d'utilisation de la tâche est consultable en exécutant la commande suivante.

```
$ php symfony help jobeet:cleanup
```

Enfin, pour l'exécuter, il suffit de l'appeler en ligne de commande tout simplement grâce à l'exécutable Symfony comme le montre le code suivant.

```
$ php symfony jobeet:cleanup --days=10 --env=dev
```

Implémenter la méthode cleanup() de la classe JobeetJobTable

La tâche n'est pas totalement fonctionnelle puisqu'il lui manque l'implémentation de la méthode `cleanup()` dans la classe `JobeetJobTable`. C'est effectivement celle-ci qui s'occupe véritablement de construire et d'appeler la bonne requête SQL qui nettoie la base de données, comme le montre le code ci-dessous.

Méthode cleanup() à ajouter au fichier 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 de Symfony se comportent parfaitement avec leur environnement puisqu'elles retournent une valeur en cas de succès. De ce fait, la valeur de retour par défaut peut être forcée en retournant explicitement un entier à la fin de la tâche.

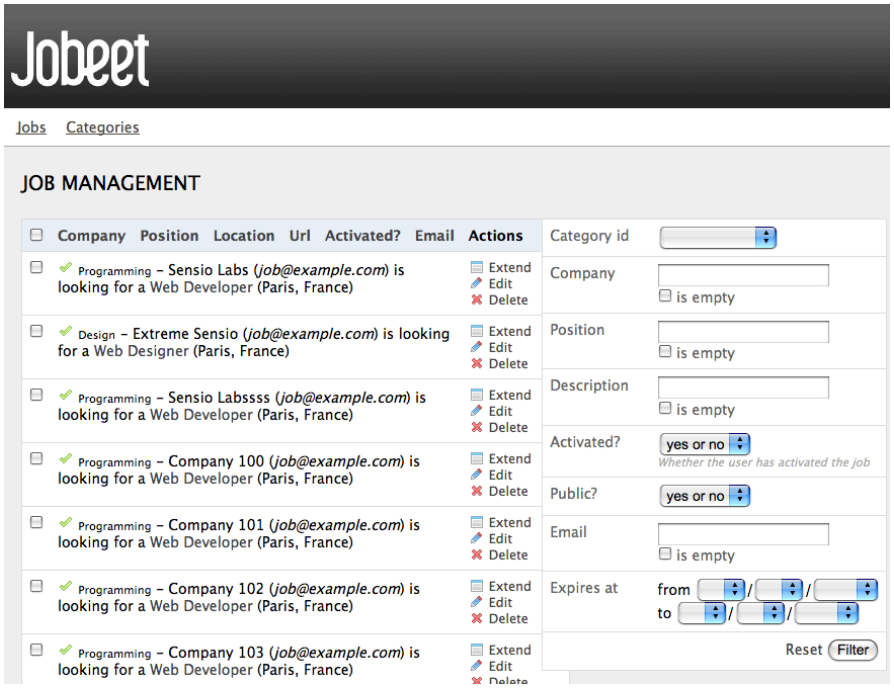
En résumé…

Tester le code est au cœur de la philosophie Symfony et de ses outils. Dans ce chapitre, nous avons vu une fois de plus comment nous servir efficacement des outils de Symfony pour rendre le processus de développement plus facile, plus rapide et surtout plus sûr.

Le framework Symfony fournit bien plus que ces quelques widgets et validateurs. Il offre en réalité une manière souple de tester les formulaires pour s'assurer qu'ils sont sécurisés par défaut.

Ce nouveau tour d'horizon des fonctionnalités de Symfony s'achève ici pour ce chapitre. Dans le chapitre suivant, il sera question de la création de l'interface d'administration dans l'application backend. Créer un back office d'administration est un passage obligé pour la plupart des projets web, et bien sûr Jobeet ne déroge pas à cette règle. La question qui se pose alors est : « Comment créer une telle interface d'administration complète et fonctionnelle en seulement quelques minutes ? ». C'est en fait extrêmement facile avec l'aide du célèbre générateur d'administration de Symfony…

chapitre 12


Jobeet
Jobs
Categories
JOB MANAGEMENT
Company
Position
Location
Url
Activated?
Email
Actions
Programming – Sensio Labs (job@example.com) is looking for a Web Developer (Paris, France)
Design – Extreme Sensio (job@example.com) is looking for a Web Designer (Paris, France)
Programming – Sensio Labssss (job@example.com) is looking for a Web Developer (Paris, France)
Programming – Company 100 (job@example.com) is looking for a Web Developer (Paris, France)
Programming – Company 101 (job@example.com) is looking for a Web Developer (Paris, France)
Programming – Company 102 (job@example.com) is looking for a Web Developer (Paris, France)
Programming – Company 103 (job@example.com) is looking for a Web Developer (Paris, France)
Extend
Edit
Delete
Category id
Company
is empty
Position
is empty
Description
is empty
Activated?
yes or no
Whether the user has activated the job
Public?
yes or no
Email
is empty
Expires at
from
to
Reset
Filter

Le générateur d'interface d'administration

MOTS-CLÉS :

- Générateur d'administration
- Fichier de configuration generator.yml
- Actions CRUD

Concevoir une interface d'administration pour une application web grand public est une tâche courante et fastidieuse à réaliser. En effet, la plupart des écrans d'un backoffice suivent le même schéma : des listes ordonnées et paginées, des formulaires de création et d'édition du contenu, des outils de recherche…

Le framework Symfony dispose d'un outil capable d'automatiser entièrement la génération de telles interfaces mais aussi d'en simplifier la configuration sans avoir (ou presque) à écrire de code PHP.

L'application développée tout au long de cet ouvrage est destinée avant tout à être alimentée par de simples utilisateurs. Tous les usagers sont libres de créer et de publier de nouvelles offres sur le site Internet. Néanmoins, cette grande liberté qui leur est offerte doit aussi être encadrée pour éviter toute dérive ou publication de contenus non conformes aux règles du site. C'est là l'un des multiples avantages de l'interface d'administration d'une application Symfony.

Avec les nouvelles fonctionnalités implémentées au chapitre précédent, l'application frontend grand public est désormais entièrement utilisable par les utilisateurs à la recherche d'emploi ou bien par ceux qui proposent de nouvelles offres. Il est donc temps de s'intéresser au développement d'une interface d'administration, développement qui sera rapidement réalisé grâce au générateur de backoffice (« admin generator ») de Symfony.

Création de l'application « backend »

Générer le squelette de l'application

La toute première étape de ce nouveau chapitre consacré à la génération du backoffice consiste à créer une nouvelle application « backend ». Cette opération a déjà été présentée au premier chapitre de cet ouvrage avec l'utilisation de la tâche `generate:app`.

```
$ php symfony generate:app --escaping-strategy=on
  ➥ --csrf-secret=UniqueSecret1 backend
```

Bien que l'application se destine uniquement aux administrateurs de Jobeet, toutes les mesures de sécurité natives ont été activées.

L'application backend est maintenant disponible à l'adresse http://jobeet.localhost/backend.php/ pour l'environnement de production, et à http://jobeet.localhost/backend_dev.php/ pour celui de développement.

ASTUCE Utiliser des caractères spéciaux pour le jeton CSRF ?

Si la valeur de l'option `--csrf-secret` contient des caractères spéciaux comme un signe $ (dollar) par exemple, ces derniers doivent être explicitement échappés à l'aide d'un antislash dans la console de lignes de commande.

```
$ php symfony generate:app
  --csrf-secret=Unique\$ecret backend
```

Recharger les jeux de données initiales

Le chargement des données initiales en base de données à partir de la tâche `doctrine:data-load` ne fonctionne plus du tout à présent. C'est en fait parce que la méthode `JobeetJob::save()` a besoin d'accéder au fichier de configuration `app.yml` de l'application `frontend`. Comme le projet est maintenant composé de deux applications, Symfony utilise le premier qu'il trouve ; en l'occurrence celui de l'application backend.

L'idéal est donc de partager le contenu du fichier de configuration de l'application frontend avec celui de l'application backoffice. Depuis l'arrivée de Symfony 1.2, cette opération est désormais rendue possible.

Le huitième chapitre a montré que les paramètres du projet pouvaient être configurés à différents niveaux. En déplaçant le fichier `apps/frontend/config/app.yml` dans le répertoire global `config/`, les paramètres de configuration seront alors partagés par l'ensemble des applications du projet. Ce changement résout le problème, et se révèle important pour la suite du chapitre dans la mesure où les classes de modèle et les variables de configuration seront sollicitées directement dans le générateur de backoffice.

ASTUCE Charger des données initiales à partir d'une configuration spécifique

La tâche `doctrine:data-load` accepte également un paramètre facultatif `--appplication` qui permet de charger les données initiales de test en utilisant la configuration propre à l'application mentionnée.

```
$ php symfony doctrine:data-load --application=frontend
```

Générer les modules d'administration

La nouvelle étape réside dans la création des modules de gestion des catégories et des offres. Bien sûr, il est hors de question de démarrer avec deux modules vides, puis de développer les différentes actions CRUD à la main pour ces deux derniers. Le générateur de backoffice de Symfony automatise toutes ces tâches afin de fournir des modules entièrement fonctionnels et personnalisables.

Générer les modules category et job

Pour l'application frontend, c'est la tâche `doctrine:generate-module` qui a été utilisée pour générer un module basique CRUD reposant sur une classe de modèle. En ce qui concerne l'application backoffice, c'est la commande `doctrine:generate-admin` qui se charge de bâtir une interface complète d'administration pour une classe de modèle donnée.

```
$ php symfony doctrine:generate-admin backend JobeetJob --module=job
$ php symfony doctrine:generate-admin backend JobeetCategory --module=category
```

Ces deux commandes créent les modules `job` et `category` pour les classes de modèle respectives `JobeetJob` et `JobeetCategory`. L'option facultative `--module` surcharge le nom du module généré par défaut par la tâche, qui aurait été `jobeet_job` pour la classe `JobeetJob`. De plus, la tâche crée aussi une collection de routes Doctrine dédiée pour chaque module.

Sans surprise, la classe de la route utilisée par le générateur d'administration est `sfDoctrineRouteCollection`, puisqu'une interface d'administration a pour but de gérer tout le cycle de vie des objets du modèle. La définition de la route déclare également quelques options vues précédemment dans cet ouvrage :

- `prefix_path` : détermine le préfixe du chemin pour la route générée. Par exemple, la page d'édition sera accessible à l'URL `/job/1/edit` ;

BONNE PRATIQUE Lire le manuel des tâches automatiques de Symfony

Le framework Symfony, et plus particulièrement les tâches automatiques, regorgent d'options et de paramètres, permettant de coller au plus près des besoins du développeur. Une bonne pratique à adopter est donc de lire la documentation de la tâche concernée, et ce, avant toute utilisation.

```
$ php symfony help
  doctrine:generate-admin
```

Le manuel présente tous les arguments et options possibles pour chaque tâche ainsi que quelques exemples pratiques simples.

- `column` : définit la colonne de la table à utiliser dans l'URL pour les liens qui référencent un objet ;
- `with_wildcard_routes` : puisque l'administration aura des opérations supplémentaires en plus des actions CRUD classiques, cette option permet de déclarer plus d'actions d'objet et de collection sans avoir à éditer la route.

Personnaliser l'interface utilisateur et l'ergonomie des modules du backoffice

Découvrir les fonctions des modules d'administration

L'exécution des deux lignes de commande précédentes a suffi à générer deux modules d'administration complets et parfaitement opérationnels. Ils sont respectivement disponibles aux URLs suivantes.

- http://jobeet.localhost/backend_dev.php/job
- http://jobeet.localhost/backend_dev.php/category

Les modules d'administration disposent de tout un tas de fonctionnalités supplémentaires en comparaison des modules auto générés traditionnels étudiés lors des chapitres précédents. Sans avoir écrit la moindre ligne de code PHP, chaque module intègre ces quelques fonctionnalités ultimes :

- la liste des objets est paginée ;
- la liste est ordonnable grâce aux en-têtes du tableau ;
- la liste peut être filtrée par le biais du formulaire de recherche sur la droite de l'écran ;
- les objets peuvent être créés, édités ou supprimés ;
- les objets sélectionnés peuvent être supprimés par lot ;
- la validation des formulaires est active ;
- les messages flash donnent immédiatement des feedbacks à l'utilisateur ;
- et bien plus encore !

Le générateur de backoffice fournit toutes les fonctionnalités nécessaires pour créer soi-même une interface de pilotage et ce, très simplement.

Améliorer le layout du backoffice

Dans le but d'améliorer l'expérience utilisateur, il est nécessaire de personnaliser davantage l'interface graphique du backend. Les deux modules sont pour l'instant accessibles individuellement par leur URL, ce qui n'est pas spécialement pratique… Le code suivant contient le contenu du fichier `layout.php` de l'application backend. Ce dernier implémente entre autres un menu de liens afin de faciliter la navigation entre les différents modules.

Contenu du layout dans le fichier 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/jobeet.gif" 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>
```

Ce layout utilise une feuille de style `admin.css` qui doit être obligatoirement présente dans le répertoire `web/css` installé au chapitre 4 avec les autres fichiers CSS.

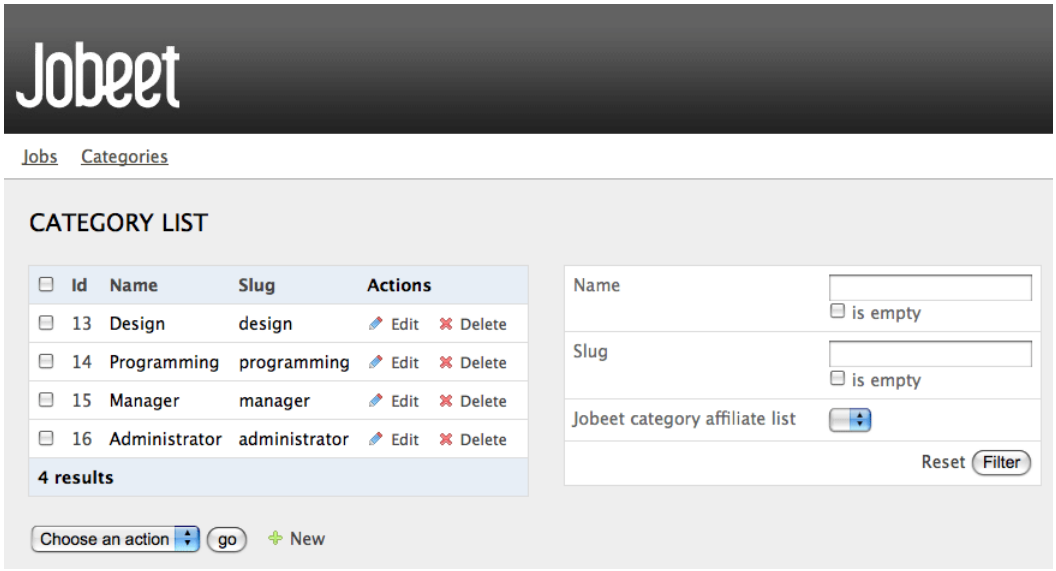


Figure 12–1
Écran d'accueil du module de gestion des catégories

Dans la foulée, l'URL de la page d'accueil du backend peut être remplacée par la liste des offres d'emploi du module `job`. Pour ce faire, il suffit de modifier la route `homepage` du fichier de configuration `routing.yml` de l'application.

Extrait du fichier apps/backend/config/routing.yml

```
homepage:
  url:   /
  param: { module: job, action: index }
```

L'étape suivante consiste à pénétrer davantage dans les entrailles du code généré pour chaque module d'administration. Il y a en effet beaucoup à apprendre sur le fonctionnement du framework Symfony, à commencer par le cache.

Comprendre le cache de Symfony

Il est temps de découvrir comment fonctionnent les modules générés, ou du moins d'étudier leur code source pour comprendre ce qu'a généré la tâche automatique pour chacun d'eux. Comme `job` et `category` sont deux modules, ils se trouvent naturellement dans le répertoire `apps/backend/modules`. En les explorant tous les deux, il est important de remarquer que les répertoires `templates/` sont tous les deux vides tandis que les fichiers d'actions le sont quasiment aussi. Le code suivant issu du fichier `actions.class.php` du module `job` en témoigne.

Contenu du fichier 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 tout cela peut-il fonctionner avec si peu de code ? En y regardant de plus près, la classe `jobActions` n'étend pas `sfActions` comme c'est le cas généralement, mais dérive la classe `autoJobActions`. Cette classe existe bel et bien dans le projet mais se trouve dans un endroit un peu inattendu. Elle appartient en réalité au répertoire `cache/backend/dev/modules/autoJob/` qui contient le véritable module d'administration.

Extrait du fichier 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())
    ))
    {

// ...
```

Le choix de générer tout le module dans le cache de Symfony n'a pas été décidé au hasard. En effet, les sections suivantes du chapitre présentent toute la force du générateur d'administration, à savoir la configuration du module grâce à un simple fichier YAML. N'importe quel changement dans ce dernier entraînera une régénération de l'intégralité du module et donc des templates et des classes. C'est pour cette raison que

les modules ne se trouvent pas dans le répertoire de l'application mais dans celui du cache.

Introduction au fichier de configuration generator.yml

La manière dont fonctionne le générateur de backoffice doit forcément rappeler quelques comportements connus. C'est en fait très similaire à ce qui a déjà été présenté au sujet des classes de formulaires et de modèles. À l'aide du schéma de description de la base de données, Symfony génère le modèle et les classes de formulaire. En ce qui concerne le générateur de backoffice, l'intégralité du module est configurable en éditant le fichier `config/generator.yml` du module. Le code suivant présente le fichier par défaut généré avec le module `job`.

Fichier de configuration 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:   1

    config:
      actions: ~
      fields: ~
      list:    ~
      filter: ~
      form:    ~
      edit:    ~
      new:     ~
```

Chaque fois que le fichier `config/generator.yml` est modifié, Symfony régénère le cache. Le renouvellement de ce dernier se produit bien sûr automatiquement en environnement de développement. En environnement de production, il doit être vidé manuellement. Les prochaines pages de ce chapitre montrent à quel point il est facile, rapide et intuitif de configurer et de personnaliser des modules construits grâce au générateur de backoffice.

Configurer les modules autogénérés par Symfony

Cette nouvelle section aborde la partie la plus importante du chapitre. Il s'agit d'apprendre comment configurer un module auto généré à partir du fichier de configuration `generator.yml`. En effet, tous les éléments qui composent les pages d'une interface d'administration sont éditables et surchargeables grâce à ce fichier.

Organisation du fichier de configuration generator.yml

Un module d'administration peut être configuré en éditant toutes les sections se trouvant sous la clé `config` du fichier de configuration `config/generator.yml`. La configuration est organisée en sept sections distinctes :

- `actions` définit la configuration par défaut des actions qui se trouvent dans la liste d'objets et dans les formulaires ;
- `fields` définit la configuration des différents champs d'un objet ;
- `list` définit la configuration de la liste des objets ;
- `filter` définit la configuration des filtres de recherche de la barre latérale de droite ;
- `form` définit la configuration des formulaires d'ajout et de modification des objets ;
- `edit` correspond à la configuration spécifique pour la page d'édition d'un objet ;
- `new` correspond à la configuration spécifique pour la page de création d'un objet.

Toutes ces sections de configuration des modules seront présentées juste après plus en détail avec des exemples concrets appliqués au backoffice de l'application. Ce processus de personnalisation démarre par la configuration des titres de chaque page du module.

Configurer les titres des pages des modules auto générés

Changer le titre des pages du module category

Pour l'instant, les titres des pages affichés au-dessus de la liste de résultats ou au-dessus des formulaires de création et d'édition sont ceux qui ont été générés par défaut par Symfony. Ils sont bien évidemment tous modifiables très simplement grâce au fichier de configuration

`generator.yml`. Il suffit en effet d'ajouter à ce dernier une sous-section `title` aux clés `list`, `new` et `edit` comme le montre le code ci-dessous.

Contenu du fichier de configuration du module category 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 titre de la section `edit` supporte des valeurs dynamiques. Les chaînes de caractères délimitées par `%%` sont remplacées par la valeur correspondante au nom de la colonne indiquée de la table. Ainsi, dans cet exemple, le motif `%%name%%` sera remplacé par le nom de la catégorie en cours de modification.

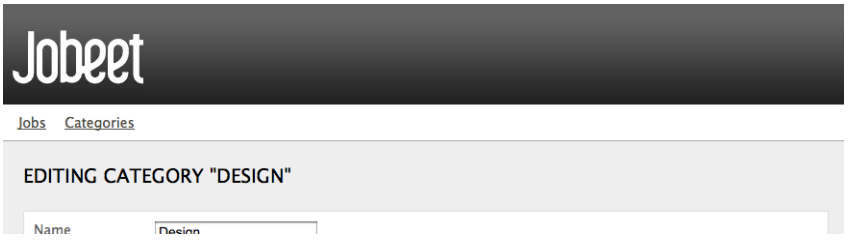


Figure 12–2
Titre dynamique de l'écran d'édition d'une catégorie

Configurer les titres des pages du module job

Dans la foulée, il est possible d'en faire autant pour le module `job` en éditant son fichier de configuration `generator.yml`.

Contenu du fichier de configuration du module job 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
```

Comme le montre le nouveau titre de la section `edit`, les paramètres dynamiques sont cumulables. Ici, le titre de la page d'édition d'une offre est composé du nom de la société (`%%company%%`) et du type de poste proposé (`%%position%%`).

Modifier le nom des champs d'une offre d'emploi

Redéfinir globalement les propriétés des champs du module

Les différentes vues (`list`, `new` et `edit`) de chaque module sont composées de « champs ». Un champ représente aussi bien le nom d'une colonne dans une classe de modèle, qu'une colonne virtuelle. Ce concept est expliqué plus loin dans ces pages. La section `fields` permet de définir globalement l'ensemble des propriétés des champs du module.

Configuration des champs du module job dans le fichier 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? }
```

Figure 12–3 Configuration des champs is_activated et is_public

Lorsqu'ils sont déclarés directement dans la section `fields`, les champs sont redéfinis globalement pour toutes les autres vues. Ainsi, le `label` du champ `is_public` sera le même quelle que soit la vue affichée : `list`, `edit` ou `new`.

Surcharger localement les propriétés des champs du module

Toutefois, il est fréquent de vouloir des intitulés différents pour chaque vue, notamment entre la page de liste et les formulaires. Toute la configuration du générateur d'administration repose sur un principe d'héritage en cascade, ce qui permet de pouvoir redéfinir certaines valeurs à

plusieurs niveaux. Par exemple, si l'on souhaite changer un `label` uniquement pour la liste d'objet, il suffit simplement de créer une nouvelle section `fields` juste en dessous de la clé `list`.

Redéfinition du label d'un champ pour la liste dans le fichier apps/backend/modules/job/config/generator.yml

```
config:
  list:
    fields:
      is_public:    { label: "Public? (label for the list)" }
```

La partie suivante explique le principe de configuration en cascade du générateur d'administration.

Comprendre le principe de configuration en cascade

N'importe quelle configuration définie sous la section principale `fields` peut être surchargée pour les besoins d'une vue spécifique. Les règles de reconfiguration sont les suivantes :

- les sections `new` et `edit` héritent de `form`, qui hérite lui-même de `fields` ;
- la section `list` hérite de `fields` ;
- la section `filter` hérite de `fields`.

Pour les sections `form` (`form`, `edit` et `new`), les options `label` et `help` surchargent celles définies dans les classes de formulaire.

Configurer la liste des objets

Les parties qui suivent expliquent l'ensemble des possibilités de configuration de la vue liste des modules auto générés. Parmi toutes ces directives de configuration figurent entre autres la déclaration des informations de l'objet à afficher, la définition du nombre d'objets par page, l'ordre par défaut de la liste, les actions par lot, etc.

Définir la liste des colonnes à afficher

Colonnes à afficher dans la liste des catégories

Par défaut, les colonnes affichées de la vue *liste* sont toutes celles du modèle, dans l'ordre du schéma de définition de la base de données. L'option `display` surcharge la configuration par défaut en définissant, dans l'ordre d'apparition, la liste des colonnes à afficher dans la liste.

Redéfinition des colonnes de la vue liste du module category dans le fichier de configuration apps/backend/modules/category/config/generator.yml

```
config:
  list:
    title:   Category Management
    display: [=name, slug]
```

Le signe = qui précède le nom de la colonne est une convention dans le framework Symfony qui permet de convertir le texte en un lien cliquable qui mène l'utilisateur au formulaire d'édition de l'objet.

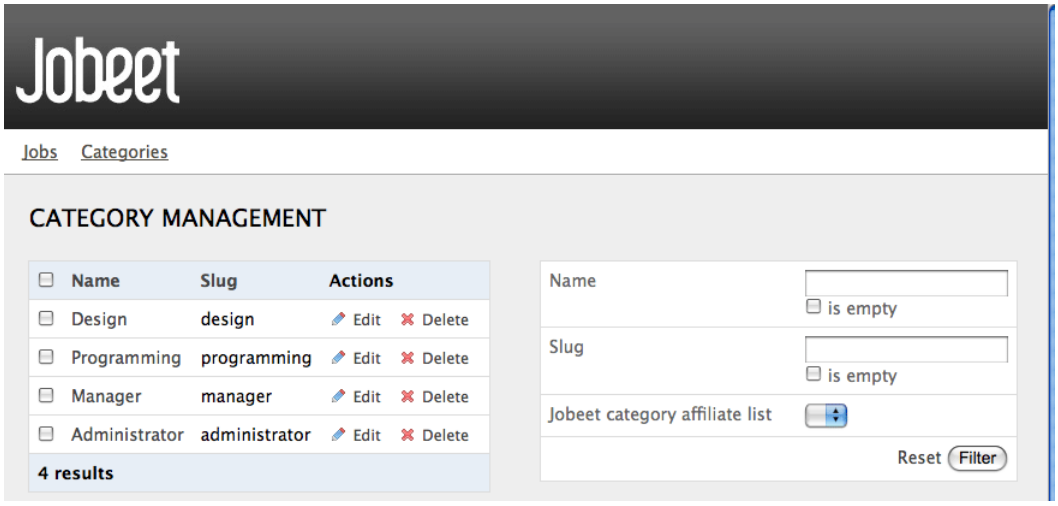


Figure 12–4
Rendu de la page d'accueil du module d'administration des catégories

Liste des colonnes à afficher dans la liste des offres

La configuration des colonnes du tableau de la vue *liste* du module `job` est exactement la même comme le démontre le code ci-après.

Définition des colonnes de la vue liste du module job dans le fichier apps/backend/modules/job/config/generator.yml

```
config:
  list:
    title:   Job Management
    display: [company, position, location, url, is_activated, email]
```

Désormais, la nouvelle liste se limite au nom de la société, le type de poste, la localisation de l'offre, l'url, le statut et l'email de contact.

Configurer le layout du tableau de la vue liste

La vue liste peut être affichée avec différents gabarits. Par défaut, c'est le layout tabulaire (`tabular`), qui signifie que chaque valeur d'une colonne

est présentée dans sa propre colonne du tableau. Il serait néanmoins plus avantageux d'avoir recours au layout linéaire (`stacked`), qui est le second gabarit disponible par défaut.

Configuration du layout stacked pour le module job dans le fichier 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%%)
```

Avec le layout linéaire, chaque objet est représenté sous la forme d'une chaîne de caractères unique, définie grâce à l'option `params`. L'option `display` reste nécessaire dans la mesure où elle détermine les colonnes grâce auxquelles l'utilisateur peut ordonner la liste des résultats.

Déclarer des colonnes « virtuelles »

Avec cette configuration, le motif `%%category_id%%` sera remplacé par la clé primaire de la catégorie à laquelle l'offre est associée. Cependant, il est beaucoup plus pertinent et significatif d'afficher le nom de la catégorie. Quelle que soit la notation `%%` utilisée, la variable n'a pas besoin de correspondre à une colonne réelle du schéma de définition de la base de données. Le générateur de backoffice doit en effet être capable de trouver un accesseur associé dans la classe de modèle.

Afin d'afficher le nom de la catégorie, il est possible de déclarer une méthode `getCategoryName()` dans la classe de modèle `JobeetJob`, puis de remplacer `%%category_id%%` par `%%category_name%%`. Or, la classe `JobeetJob` dispose déjà d'une méthode `getJobeetCategory()` qui retourne l'objet catégorie associé. De ce fait, en utilisant le motif `%%jobeet_category%%`, le nom de la catégorie s'affiche dans la liste d'offres d'emploi car la classe `JobeetCategory` implémente la méthode magique `__toString()` qui convertit l'objet en une chaîne de caractères.

Code à remplacer dans le fichier apps/backend/modules/job/config/generator.yml

```
%%is_activated%% <small>%%jobeet_category%%</small> -
%%company%%
(<em>%%email%%</em>) is looking for a %%=position%%
(%%location%%)
```

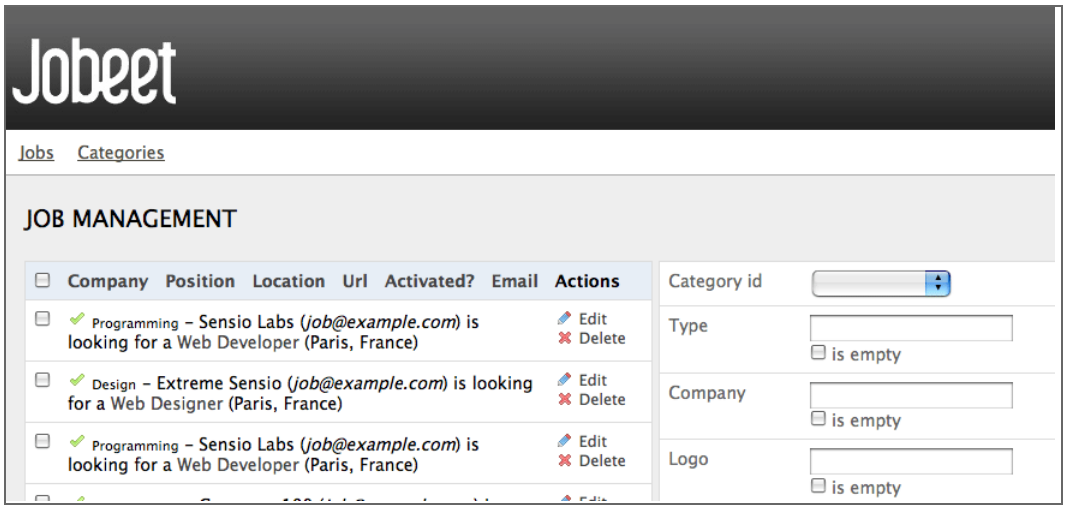


Figure 12–5
Ajout de la catégorie pour chaque objet

Définir le tri par défaut de la liste d'objets

Un administrateur préférera certainement voir les dernières offres d'emploi postées sur la première page. L'ordre des enregistrements dans la liste est configurable grâce à l'option `sort` de la section `list` comme le montre le code ci-dessous.

Définition de l'ordre des objets dans le tableau de la vue liste dans le fichier apps/backend/modules/job/config/generator.yml

```
config:
  list:
    sort: [expires_at, desc]
```

La première valeur du tableau `sort` correspond au nom de la colonne sur laquelle le tri effectué, tandis que la seconde définit le sens. La valeur `desc` détermine un ordre descendant, c'est-à-dire du plus grand au plus petit (ou de Z à A pour les chaînes, ou bien du plus récent au plus vieux avec les dates). Pour obtenir un ordre ascendant (par défaut), il suffit de d'indiquer la valeur `asc`.

Réduire le nombre de résultats par page

Par défaut, la liste est paginée et chaque page contient 20 enregistrements. Cette valeur est bien sûr éditable grâce à l'option `max_per_page` de la section `list`.

Définition du nombre d'enregistrements par page dans le fichier apps/backend/modules/job/config/generator.yml

```
config:
  list:
    max_per_page: 10
```

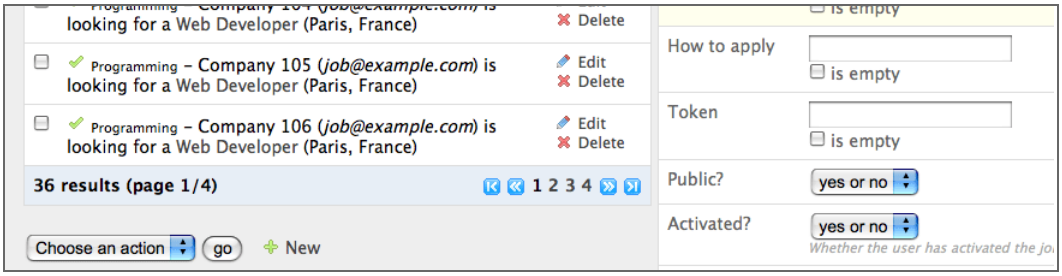


Figure 12–6
Pagination de la liste d'objets

Configurer les actions de lot d'objets

Le générateur de backoffice de Symfony intègre nativement la possibilité d'exécuter des actions sur un lot d'objets sélectionnés dans le tableau de la vue *liste*. Le gestionnaire n'en a pas véritablement besoin, c'est pourquoi la première partie explique comment les désactiver. En revanche, la seconde partie présente comment configurer de nouvelles actions de lot pour le module `job`.

Désactiver les actions par lot dans le module category

La vue *liste* permet d'exécuter une action sur plusieurs objets en même temps. Ces actions par lot ne sont pas nécessaires pour le module `category`, c'est pourquoi le code montre la manière de les retirer à l'aide d'une simple configuration du fichier `generator.yml`.

Suppression des actions de lot dans le fichier apps/backend/modules/category/config/generator.yml

```
config:
  list:
    batch_actions: {}
```

L'option `batch_actions` définit la liste des actions applicables sur un lot d'objets Doctrine. Indiquer explicitement un tableau vide en guise de valeur permet de supprimer cette fonctionnalité.

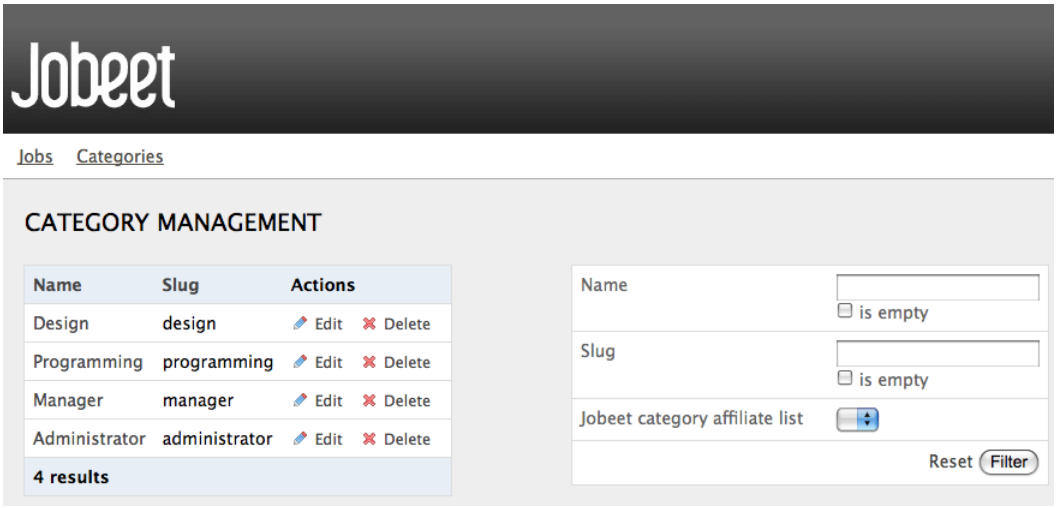


Figure 12–7
Liste des catégories après suppression des actions par lot d'objets

Ajouter de nouvelles actions par lot dans le module job

Par défaut, chaque module dispose d'une action de suppression par lot générée automatiquement par le framework. En ce qui concerne le module `job`, il serait utile de pouvoir étendre la validité de quelques objets sélectionnés pour 30 jours supplémentaires, grâce à une nouvelle action.

Ajout de la nouvelle action extends dans le fichier apps/backend/modules/job/config/generator.yml

```
config:
  list:
    batch_actions:
      _delete:    ~
      extend:     ~
```

Toutes les actions commençant par un tiret souligné (underscore) sont des actions natives fournies par le framework. En rafraîchissant le navigateur, la liste déroulante accueille à présent l'action *extend* mais Symfony lance une exception indiquant qu'une nouvelle méthode `executeBatchExtend()` doit être créée.

Méthode executeBatchExtend() à ajouter au fichier 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');
  }
}
```

Bien que la compréhension de ce code ne pose pas de difficulté particulière, quelques explications supplémentaires s'imposent. La liste des clés primaires des objets sélectionnés est stockée dans le paramètre `ids` de l'objet de requête. Ce tableau d'identifiants uniques a été transmis en POST lors de la soumission du formulaire. La variable `$ids` est ensuite transmise à la requête Doctrine qui se charge de récupérer et d'hydrater tous les objets correspondant à cette liste d'identifiants.

L'appel à la méthode `execute()` sur l'objet `Doctrine_Query` retourne un objet `Doctrine_Collection` contenant toutes les offres `JobeetJob` correspondantes récupérées dans la base de données. Pour chaque offre d'emploi, l'appel à la méthode `extend()` permet de prolonger la durée de vie de l'objet pour une durée de 30 jours supplémentaires.

Enfin, un nouveau message de feedback est écrit dans la session de l'utilisateur afin de lui informer que les offres sélectionnées ont bien été prolongées après sa redirection.

Le paramètre `true` de la méthode `extend()` permet de contourner la vérification de la date d'expiration. Le code suivant implémente cette nouvelle fonctionnalité.

Édition de la méthode extend() de la classe JobeetJob dans le fichier 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;
  }

  // ...
}
```

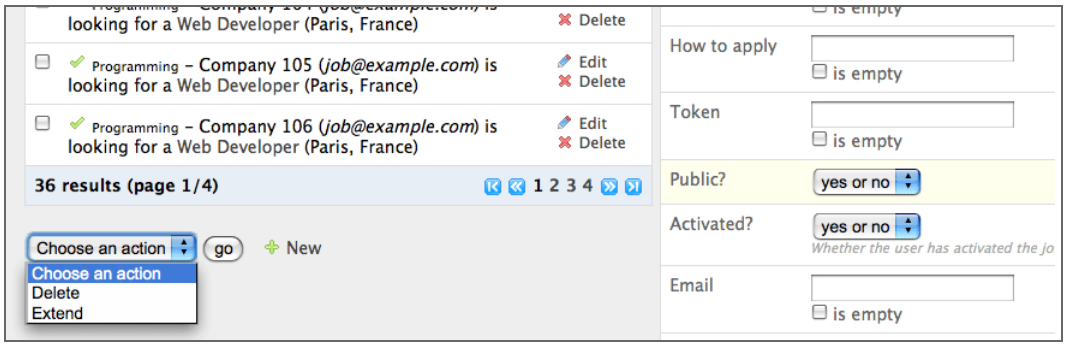


Figure 12–8
Ajout de l'action extend à la liste déroulante des actions de lot

Configurer les actions unitaires pour chaque objet

Le tableau de la vue *liste* contient une colonne additionnelle destinée aux actions applicables unitairement sur chaque objet. Par défaut, Symfony introduit les actions d'édition et de suppression d'un enregistrement, mais il est évidemment possible d'en ajouter de nouvelles en éditant le fichier de configuration du module.

Supprimer les actions d'objets des catégories

En considérant que le lien sur le titre de la catégorie suffise pour accéder au formulaire d'édition et que la suppression d'une catégorie soit interdite, les actions d'objet n'ont plus véritablement de raison de persister. La configuration suivante retire complètement la dernière colonne du tableau de catégories.

Suppression des actions d'objets dans le fichier apps/backend/modules/category/config/generator.yml

```
config:
  list:
    object_actions: {}
```

De la même manière que pour les actions par lot, spécifier un tableau vide comme valeur à l'option `object_actions` permet de retirer les actions des objets du tableau.

Ajouter d'autres actions pour chaque offre d'emploi

Pour le module `job`, il convient de conserver les actions d'édition et de suppression pour chaque item du tableau. Le code ci-dessous montre comment ajouter une nouvelle action *extend* pour chaque objet. Cette dernière permet d'étendre la durée de vie d'une offre de manière unitaire au simple clic sur le lien créé par Symfony.

Ajout de la nouvelle action d'objet extend au fichier apps/backend/modules/job/config/generator.yml

```
config:
  list:
    object_actions:
      extend:     ~
      _edit:      ~
      _delete:    ~
```

Pour fonctionner complètement, cette action doit être accompagnée de la déclaration d'une méthode `executeListExtend()` dans la classe d'actions. Celle-ci doit implémenter la logique nécessaire à la prolongation de la durée de vie d'une offre comme le présente le code suivant.

Ajout de la méthode executeListExtend() au fichier 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');
  }

  // ...
}
```

Configurer les actions globales de la vue liste

Pour l'instant, il est possible de créer des liens vers des actions pour la liste entière ou bien pour un seul enregistrement du tableau. L'option `actions` définit la liste des actions qui ne sont pas directement en relation avec les objets, c'est le cas par exemple de la création d'un nouvel objet.

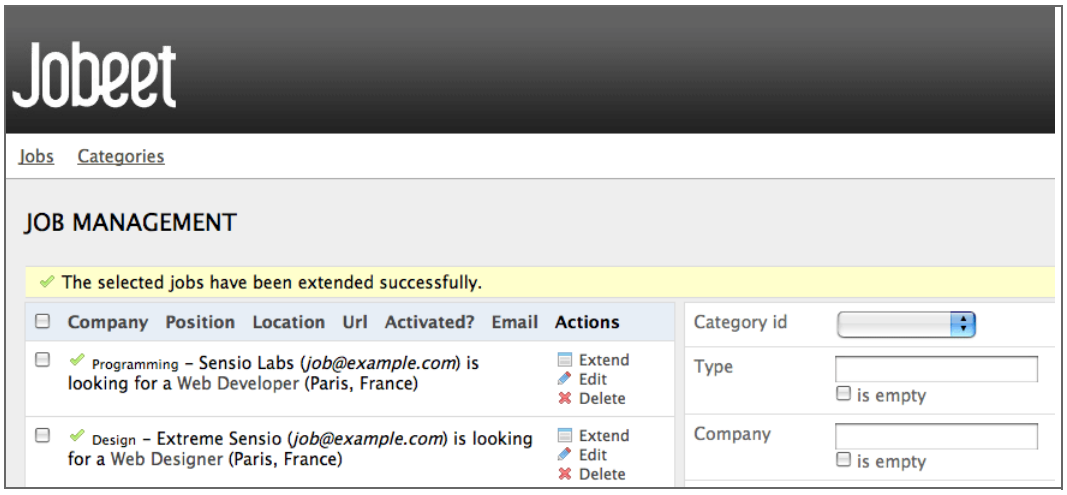


Figure 12–9
Ajout de l'action extend pour chaque objet

Dans Jobeet, ce sont les utilisateurs qui postent de nouvelles offres. Les administrateurs n'en ont pas la nécessité, c'est pourquoi le lien de création d'un nouvel objet peut être supprimé. En revanche, les administrateurs doivent avoir la possibilité de purger la base de données des offres expirées de plus de 60 jours.

Ajout d'une nouvelle action globale deleteNeverActivated dans le fichier apps/backend/modules/job/config/generator.yml

```
config:
  list:
    actions:
      deleteNeverActivated: { label: Delete never activated jobs }
```

Jusqu'à maintenant, toutes les actions globales de la vue *liste* étaient déclarées avec le symbole « tilde » (~), ce qui signifie que Symfony configure l'action automatiquement. Chaque action peut être personnalisée en définissant un tableau de paramètres. L'option `label` surcharge l'intitulé automatiquement généré par le framework. Par défaut, l'action exécutée lorsque l'on clique sur le lien est le nom de l'action préfixé par `list`. Le code ci-dessous montre l'implémentation de l'action globale `deleteNeverActivated` pour le module `job`.

Implémentation de la méthode listDeleteNeverActivated dans le fichier 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');
  }

  // ...
}
```

Cette action n'est guère complexe à comprendre. La première ligne du corps de la méthode fait appel à la méthode `cleanup()` qui se charge de supprimer de la base de données toutes les offres expirées depuis plus de 60 jours. Cette méthode a été implémentée lors du précédent chapitre. Elle prend en paramètre le nombre de jours succédant la date d'expiration de l'offre et retourne le nombre d'enregistrements effacés de la base de données. Enfin, en fonction du nombre d'objets effacés, un message flash est fixé dans la session de l'utilisateur avant de le rediriger vers la page d'accueil du module d'administration.

La réutilisation de la méthode `cleanup()` est ici un exemple particulièrement démonstratif des avantages du motif de conception MVC.

Le nom de la méthode à écrire risquera d'être trop peu explicite et fastidieux à écrire. Dans ce cas, il est possible de remplacer le nom généré par défaut.

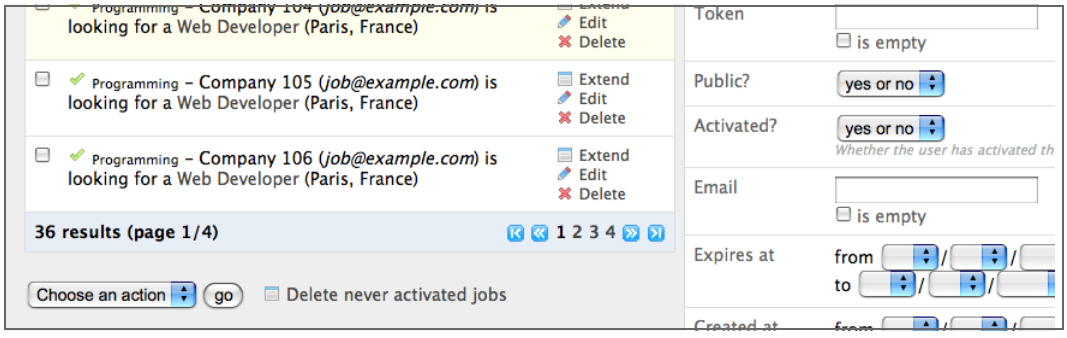


Figure 12–10
Ajout de fonctionnalités globales à la vue liste

Le nom de l'action à utiliser peut aussi être redéfini afin de le simplifier. Il suffit alors d'ajouter un paramètre `action` dans la déclaration de l'action globale comme le présente le code ci-dessous.

```
deleteNeverActivated: { label: Delete never activated jobs,
action: foo }
```

Optimiser les requêtes SQL de récupération des enregistrements

En affichant l'onglet *SQL* de la barre de déboguage de Symfony, on constate ici que la liste a besoin d'exécuter 14 requêtes SQL pour afficher la liste des offres d'emploi. Or, parmi ces 14 requêtes, il y en a 10 qui remplissent le même besoin : récupérer le nom de la catégorie associée à l'enregistrement. Cet exemple sous-entend clairement le manque d'une jointure entre les relations `jobeet_job` et `jobeet_category` qui permettrait de sélectionner à la fois les offres et leur catégorie respective à l'aide d'une seule et même requête SQL.

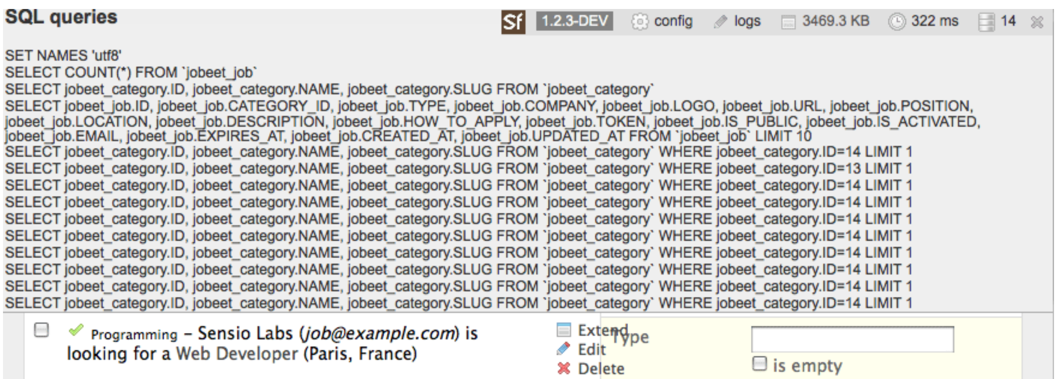


Figure 12–11
Détail des requêtes SQL générées pour afficher la liste d'objets

Par défaut, le générateur de backoffice utilise la méthode Doctrine la plus simple pour récupérer une liste d'enregistrements. De ce fait, lorsque des objets sont en relation, l'ORM est obligé d'effectuer les requêtes adéquates pour les retrouver et hydrater l'objet associé à la demande.

Pour éviter ce comportement et cette perte de performance, l'option `table_method` permet de surcharger la méthode Doctrine utilisée par défaut par le framework pour générer la liste de résultats. Dès lors, il est possible de déclarer une nouvelle méthode dans le modèle qui implémente la jointure entre les deux tables.

Les deux listings de code suivants expliquent comment configurer l'emploi d'une nouvelle méthode du modèle pour la récupération des

enregistrements et de quelle manière elle doit être implémentée pour le module `job`.

Surcharge de la méthode de récupération des enregistrements dans le fichier apps/backend/modules/job/config/generator.yml

```
config:
  list:
    table_method: retrieveBackendJobList
```

Il ne reste alors plus qu'à implémenter cette nouvelle méthode `retrieveBackendJobList` dans la classe `JobeetJobTable` qui se trouve dans le fichier `lib/model/doctrine/JobeetJobTable.class.php`.

Implémentation de la méthode retrieveBackendJobList dans le fichier 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;
  }

  // ...
```

Cette méthode `retrieveBackendJobList()` reçoit un objet `Doctrine_Query` en paramètre auquel elle ajoute la condition de jointure entre les tables `jobeet_categoy` et `jobeet_job` dans le but de créer automatiquement chaque objet `JobeetJob` et `JobeetCategory`. Maintenant, grâce à cette requête optimisée, le nombre de requêtes SQL exécutées pour générer la vue *liste* tombe à 4 comme l'atteste l'onglet *SQL* de la barre de déboguage de Symfony.

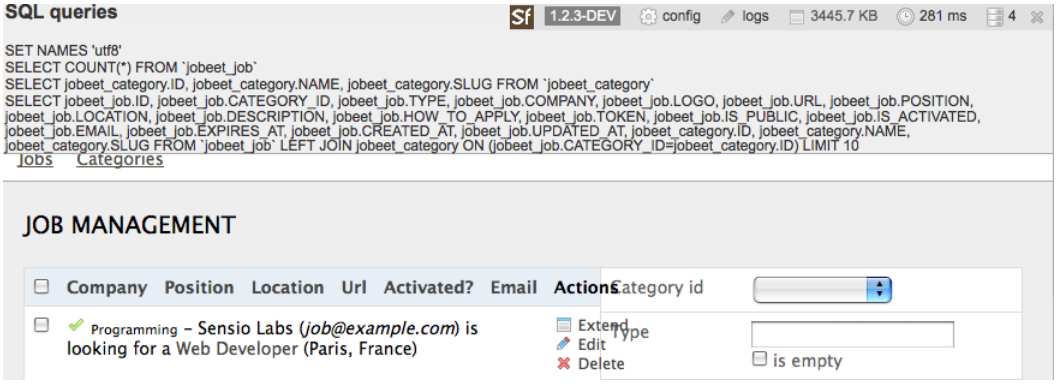


Figure 12–12
Détail des requêtes SQL générées pour afficher la liste d'objets après optimisation

La configuration de la vue *list* des modules `category` et `job` s'achève ici. Les deux modules disposent à présent chacun d'une vue *list* entièrement fonctionnelle, paginée et adaptée aux besoins des administrateurs de l'application. Il est désormais temps de s'intéresser à la personnalisation des formulaires qui composent les vues *new* et *edit*.

Configurer les formulaires des vues de saisie de données

La configuration des vues de formulaires se décompose en trois sections distinctes dans le fichier `generator.yml` : `form`, `edit` et `new`. Toutes possèdent les mêmes possibilités de configuration dans la mesure où la section `form` peut être surchargée dans les deux autres sections.

Les parties qui suivent expliquent comment configurer les formulaires qui permettent de créer et d'éditer les objets Doctrine en vue de leur sérialisation dans la base de données. La configuration de ces vues intervient à différents niveaux tels que l'ajout ou la suppression de champs, la modification de leurs propriétés respectives, le choix d'une classe de formulaire personnalisée à la place de celle par défaut…

Configurer la liste des champs à afficher dans les formulaires des offres

De la même manière que pour la vue *list*, il est possible de changer le nombre et l'ordre des champs affichés dans les formulaires grâce à l'option `display`. Comme le formulaire affiché est défini par une classe, il est recommandé de ne pas tenter de supprimer de champ dans la mesure où cela risque de conduire à des erreurs de validation inattendues. Le code ci-dessous explique comment utiliser l'option `display` pour configurer la liste des champs à afficher dans le formulaire. Cette option a la particularité de faciliter l'ordonnancement des champs par groupes d'information de même nature.

Configuration des formulaires dans le fichier 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 d'informations (`Content` et `Admin`), dont chacun contient un sous-ensemble des champs du formulaire. La capture d'écran ci-dessous montre le rendu obtenu à partir de cette configuration du formulaire.

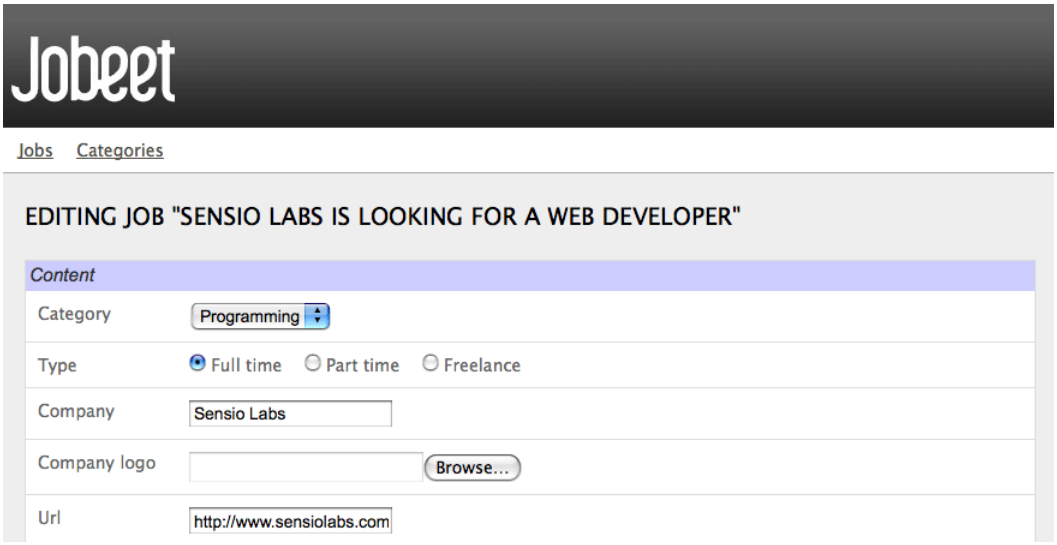


Figure 12–13
Rendu du formulaire d'édition d'une offre d'emploi

Les colonnes du groupe d'informations `Admin` ne s'affichent pas encore dans le navigateur car elles ont été retirées de la définition du formulaire de gestion d'une offre. Elles apparaîtront plus tard dans ce chapitre lorsqu'une classe de formulaire d'offre d'emploi personnalisée sera définie pour l'application backoffice.

Le générateur d'administration dispose d'un support natif pour les relations plusieurs à plusieurs (*many to many*). Sur le formulaire de manipulation d'une catégorie, il existe un champ pour le nom et pour le slug, ainsi qu'une liste déroulante pour les partenaires associés. Éditer cette relation dans cette page n'a pas véritablement de sens, c'est pourquoi elle peut être retirée du formulaire comme le montre la configuration de la classe de formulaire ci-dessous.

Configuration de la liste des champs du formulaire de catégorie dans le fichier lib/form/doctrine/JobeetCategoryForm.class.php

```
class JobeetCategoryForm extends BaseJobeetCategoryForm
{
  public function configure()
  {
    unset($this['created_at'], $this['updated_at'],
$this['jobeet_affiliates_list']);
  }
}
```

La section suivante aborde la notion de champs virtuels, c'est-à-dire des champs supplémentaires qui n'appartiennent pas à la définition de base des widgets de la classe de formulaire.

Ajouter des champs virtuels au formulaire

Dans l'option `display` du formulaire d'offre d'emploi figure le champ `_generated_token` dont le nom commence par un underscore. Cela signifie que le rendu de ce champ est pris en charge par un template partiel nommé `_generated_token.php`. Le contenu de ce fichier à créer est présenté dans le bloc de code ci-dessous.

Contenu du template partiel dans le fichier apps/backend/modules/job/templates/_generated_token.php

```
<div class="sf_admin_form_row">
  <label>Token</label>
  <?php echo $form->getObject()->getToken() ?>
</div>
```

Un partial a accès au formulaire courant (`$form`), et donc à l'objet associé via la méthode `getObject()` appliquée sur cet objet de formulaire. Le rendu du champ virtuel peut aussi être délégué à un composant en préfixant son nom par un tilde.

Redéfinir la classe de configuration du formulaire

Comme le formulaire sera manipulé par les administrateurs, quelques informations nouvelles ont été ajoutées en complément par rapport au formulaire de l'application frontend. Pour l'instant, certaines d'entre elles n'apparaissent pas sur le formulaire puisqu'elles ont été supprimées dans la classe `JobeetJobForm`.

Implémenter une nouvelle classe de formulaire par défaut

Afin d'obtenir différents formulaires entre les applications frontend et backend, il est nécessaire de créer deux classes séparées, dont une intitulée `BackendJobeetJobForm` qui étend la classe `JobeetJobForm`. Comme les champs cachés ne sont pas les mêmes dans les deux formulaires, la classe `JobeetJobForm` doit être remaniée légèrement afin de déplacer l'instruction `unset()` dans une méthode qui sera redéfinie dans `BackendJobeetJobForm`.

Contenu de la classe JobeetJobForm dans le fichier lib/form/doctrine/JobeetJobForm.class.php

```
class JobeetJobForm extends BaseJobeetJobForm
{
  public function configure()
  {
    $this->removeFields();

    $this->validatorSchema['email'] = new sfValidatorEmail();

    // ...
  }

  protected function removeFields()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['expires_at'], $this['is_activated'],
      $this['token']
    );
  }
}
```

Contenu de la classe BackendJobeetJobForm dans le fichier lib/form/doctrine/BackendJobeetJobForm.class.php

```
class BackendJobeetJobForm extends JobeetJobForm
{
  public function configure()
  {
    parent::configure();
  }

  protected function removeFields()
  {
    unset(
      $this['created_at'], $this['updated_at'],
      $this['token']
    );
  }
}
```

À présent, la classe de formulaire par défaut utilisée par le générateur d'administration peut être surchargée grâce au paramètre `class` du fichier de configuration `generator.yml`. Avant de rafraîchir le navigateur, le cache de Symfony doit être vidé afin de prendre en compte la nouvelle classe créée dans le fichier d'auto chargement de classes.

Modification du nom de la classe par défaut pour les formulaires de l'application backend dans le fichier apps/backend/modules/job/config/generator.yml

```
config:
  form:
    class: BackendJobeetJobForm
```

Le formulaire de modification possède néanmoins un léger inconvénient. La photo courante téléchargée de l'objet n'est affichée nulle part sur le formulaire, et il est impossible de forcer la suppression de l'actuelle.

Implémenter un meilleur mécanisme de gestion des photos des offres

Le widget `sfWidgetFormInputFileEditable` apporte des capacités supplémentaires d'édition de fichier par rapport au widget classique de téléchargement de fichier. Le code suivant remplace le widget actuel du champ logo par un widget de type `sfWidgetFormInputFileEditable`, afin de permettre aux administrateurs de faciliter la gestion des images téléchargées pour chaque offre.

Modification du widget du champ logo dans le 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` prend plusieurs options en paramètres pour personnaliser ses fonctionnalités et son rendu dans le navigateur :

- `file_src` détermine le chemin web vers le fichier courant ;
- `is_image` indique si oui ou non le fichier doit être affiché comme une image ;

- `edit_mode` spécifie si le formulaire est en mode d'édition ou s'il ne l'est pas ;
- `with_delete` permet d'ajouter ou non une case à cocher pour forcer la suppression du fichier ;
- `template` définit le gabarit HTML pour le rendu du widget.

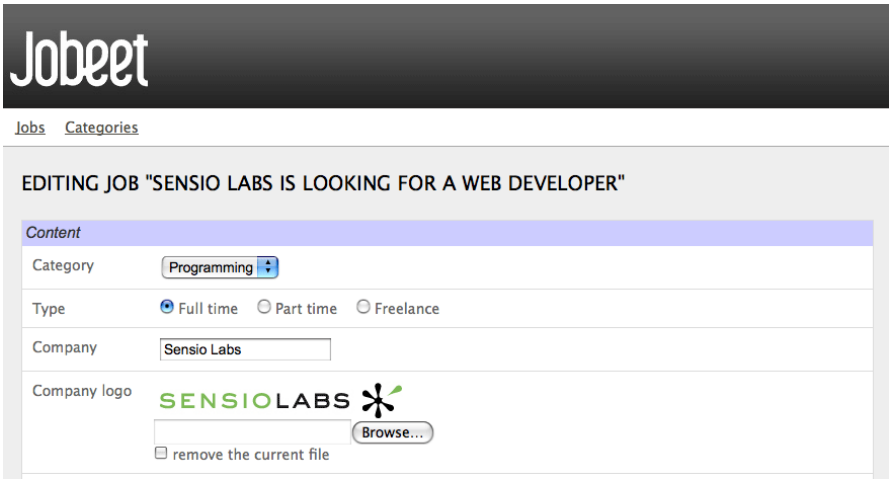


Figure 12–14 Rendu du widget de modification de téléchargement de fichier

L'apparence du générateur de backoffice peut facilement être personnalisée dans la mesure où les templates générés définissent un nombre important de classes CSS et d'attributs `id`. Par exemple, le champ logo peut être mis en forme en utilisant la classe CSS `sf_admin_form_field_logo`. Chaque champ du formulaire dispose de sa propre classe relative au type de widget, telles que `sf_admin_text` ou `sf_admin_boolean`.

L'option `edit_mode` utilise la méthode `isNew()` de l'objet `sfDoctrineRecord`. Elle retourne `true` si l'objet modèle du formulaire est nouveau (création) et `false` dans le cas contraire (édition). Cette méthode est une aide indéniable lorsque le formulaire requiert des widgets ou des validateurs qui dépendent du statut de l'objet embarqué.

Configurer les filtres de recherche de la vue liste

Configurer les filtres de recherche est sensiblement similaire à configurer les vues de formulaire. En réalité, les filtres sont tout simplement des formulaires. Ainsi, comme avec les formulaires, les classes de filtre ont été automatiquement générées par la tâche `doctrine:build-all`, mais peuvent également être reconstruites à l'aide de la tâche `doctrine:build-filters`.

Les classes des formulaires de filtre sont situées dans le répertoire `lib/filter`, et chaque classe de modèle dispose de sa propre classe de filtre (`JobeetJobFormFilter` pour `JobeetJob`).

Supprimer les filtres du module de category

Comme le module de gestion des catégories ne nécessite guère de filtres, ces derniers peuvent être désactivés dans le fichier de configuration `generator.yml` comme le montre le code ci-dessous.

Suppression de la liste de filtres du module category dans le fichier apps/backend/modules/category/config/generator.yml

```
config:
  filter:
    class: false
```

Configurer la liste des filtres du module job

Par défaut, la liste de filtres permet de réduire la sélection des objets en agissant sur toutes les colonnes de la table. Or, tous les filtres ne sont pas pertinents, c'est pourquoi certains d'entre eux peuvent être retirés pour les offres d'emploi.

Modification de la liste de filtres du module job dans le fichier apps/backend/modules/job/config/generator.yml

```
filter:
  display: [category_id, company, position, description,
is_activated, is_public, email, expires_at]
```

Tous les filtres sont optionnels. De ce fait, il n'y a aucun besoin de surcharger la classe de formulaire de filtres pour configurer les champs à afficher.

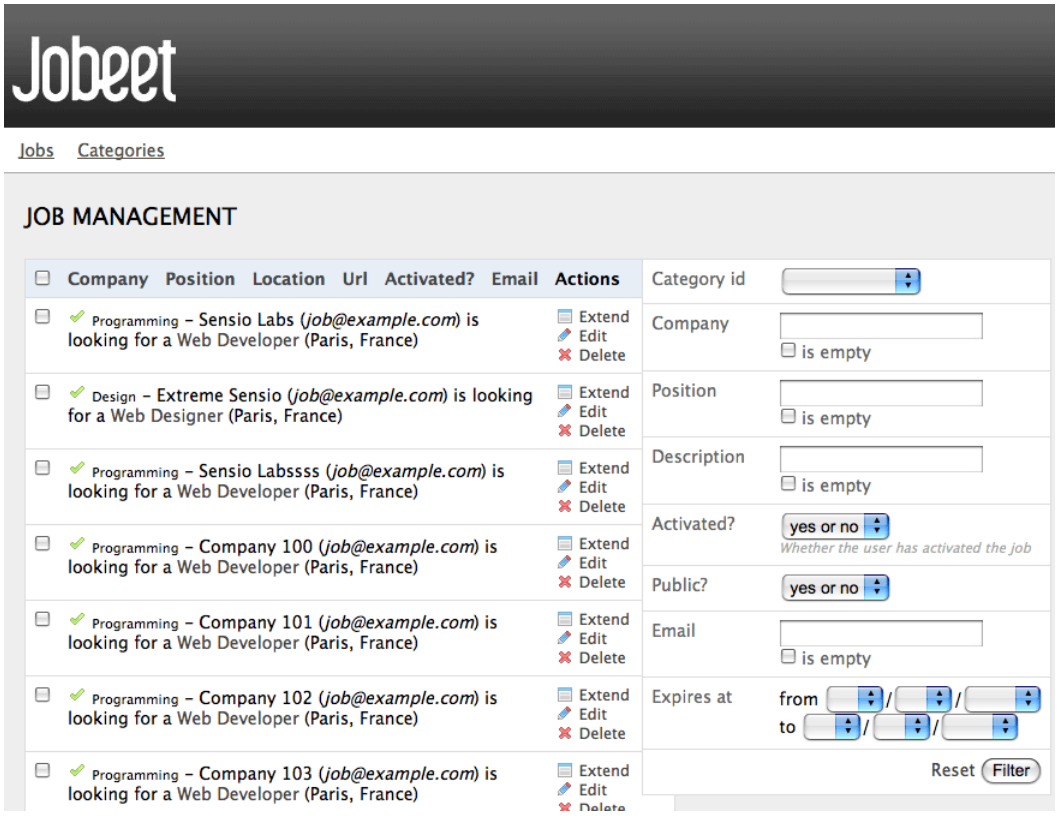


Figure 12–15
Rendu de la liste des filtres après configuration

Personnaliser les actions d'un module autogénéré

Lorsque la configuration ne suffit plus, il est toujours possible d'ajouter de nouvelles méthodes à la classe des actions, comme cela a déjà été expliqué avec la prolongation de la durée de vie d'une offre d'emploi. Par ailleurs, toutes les actions autogénérées par le générateur de backoffice peuvent être redéfinies grâce à l'héritage de classe. Le tableau ci-dessous dresse la liste intégrale de ces méthodes autogénérées dont on peut surcharger la configuration depuis la classe d'actions du module.

Tableau 12–1 Liste des actions auto générées par le générateur d'administration

Nom de la méthode	Description
`executeIndex()`	Exécute la vue `list`
`executeFilter()`	Met à jour les filtres
`executeNew()`	Exécute l'action de la vue `new`

Tableau 12–1 Liste des actions auto générées par le générateur d'administration (suite)

Nom de la méthode	Description
`executeCreate()`	Crée une nouvelle offre dans la base de données
`executeEdit()`	Exécute l'action de la vue `edit`
`executeUpdate()`	Met à jour une offre dans la base de données
`executeDelete()`	Supprime une offre d'emploi
`executeBatch()`	Exécute une action sur un lot d'objets
`executeBatchDelete()`	Exécute l'action de suppression par lot `_delete`
`processForm()`	Traite le formulaire d'une offre d'emploi
`getFilters()`	Retourne la liste des filtres courants
`setFilters()`	Définit les filtres courants
`getPager()`	Retourne l'objet de pagination de la vue liste
`getPage()`	Retourne la page courante de pagination
`setPage()`	Définit la page courante de pagination
`buildCriteria()`	Construit le critère de tri de la liste
`addSortCriteria()`	Ajoute le critère de tri à la liste
`getSort()`	Retourne la colonne de tri courante
`setSort()`	Définit la colonne de tri courant

Il faut savoir que chaque méthode générée ne réalise qu'un traitement bien spécifique, ce qui permet de modifier certains comportements sans avoir à copier et coller trop de code, ce qui irait à l'encontre de la philosophie DRY du framework.

Personnaliser les templates d'un module autogénéré

Tous les templates construits par le générateur d'administration ont la particularité d'être facilement personnalisables dans la mesure où ils contiennent tous de nombreuses classes CSS et attributs `id` dans le code HTML. Cependant, il arrive parfois que cela ne suffise pas pour aboutir au même résultat que la maquette graphique choisie par le client.

Au même titre que pour les classes, tous les templates originaux des modules autogénérés sont entièrement redéfinissables. Il est important de rappeler qu'un template n'est en fait qu'un simple fichier PHP contenant uniquement du code HTML et PHP. Créer un nouveau template du même nom que l'original dans le répertoire `apps/backend/modules/job/templates/` (où job est ici le nom du module concerné) suffit pour qu'il soit utilisé.

Le tableau suivant décrit l'ensemble des templates nécessaires au bon fonctionnement des modules bâtis à partir du générateur de backoffice. Tous remplissent une fonction bien précise dans la génération d'une page d'administration, ce qui permet, comme pour les classes autogénérées, de pouvoir en surcharger seulement quelques uns pour personnaliser le rendu d'un module.

Tableau 12–2 Liste des templates autogénérés par le générateur d'administration

Nom du template	Description
`_assets.php`	Rend les feuilles de style et JavaScript à utiliser dans le template
`_filters.php`	Rend la barre latérale des filtres
`_filters_field.php`	Rend un champ unique de la barre de filtres
`_flashes.php`	Rend les messages flash de feedback
`_form.php`	Affiche le formulaire
`_form_actions.php`	Affiche les actions du formulaire
`_form_field.php`	Affiche un champ unique du formulaire
`_form_fieldset.php`	Affiche un groupe de champs de même nature dans le formulaire
`_form_footer.php`	Affiche le pied du formulaire
`_form_header.php`	Affiche l'en-tête du formulaire
`_list.php`	Affiche la liste d'objets
`_list_actions.php`	Affiche les actions de la liste
`_list_batch_actions.php`	Affiche les actions de lot d'objets de la liste
`_list_field_boolean.php`	Affiche un champ de type booléen dans la liste
`_list_footer.php`	Affiche le pied de la liste
`_list_header.php`	Affiche l'en-tête de la liste
`_list_td_actions.php`	Affiche les actions unitaires d'un objet représenté par une ligne du tableau
`_list_td_batch_actions.php`	Affiche la case à cocher d'un objet
`_list_td_stacked.php`	Affiche le layout *stacked* d'une ligne
`_list_td_tabular.php`	Affiche un champ unique d'une ligne
`_list_th_stacked.php`	Affiche les propriétés d'un enregistrement dans une seule colonne sur toute la ligne
`_list_th_tabular.php`	Affiche les propriétés d'un enregistrement dans des colonnes séparées du tableau
`_pagination.php`	Affiche la pagination de la liste d'objets
`editSuccess.php`	Affiche la vue d'édition d'un objet
`indexSuccess.php`	Affiche la vue liste du module
`newSuccess.php`	Affiche la vue de création d'un nouvel objet

La configuration finale du module

Avec seulement deux fichiers de configuration et quelques ajustements dans le code PHP générés, l'application Jobeet se dote d'une interface d'administration complète en un temps record. Les deux codes suivants synthétisent toute la configuration finale des modules présentée pas à pas tout au long de ce chapitre.

Configuration finale du module job

Configuration finale du module job dans le fichier 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:   1

    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
```

Configuration finale du module category

Configuration finale du module category dans le fichier 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:   1

    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
```

ASTUCE Configuration YAML vs configuration PHP

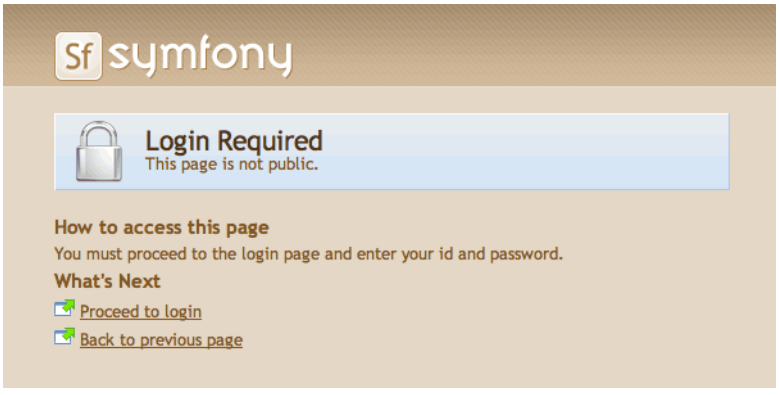
À présent, vous savez que lorsque quelque chose est configurable dans un fichier YAML, c'est également possible avec du code PHP pur. En ce qui concerne le générateur de backoffice, toute la configuration PHP se trouve dans le fichier `apps/backend/modules/job/lib/jobGeneratorConfiguration.class.php`. Ce dernier fournit les mêmes options que le fichier YAML mais avec une interface en PHP. Afin d'apprendre et de connaître les noms des méthodes de configuration, il suffit de jeter un oeil aux classes de base auto-générées (par exemple `BaseJobGeneratorConfiguration`) dans le fichier de configuration PHP
`cache/backend/dev/modules/autoJob/lib/`
➥ `BaseJobGeneratorConfiguration.class.php`.

En résumé…

En seulement moins d'une heure, le projet Jobeet dispose d'une interface complète d'administration des catégories et des offres d'emploi déposées par les utilisateurs. Mais le plus étonnant, c'est que l'écriture de toute cette interface de gestion n'aura même pas nécessité plus de cinquante lignes de code PHP. Ce n'est pas si mal pour autant de fonctionnalités intégrées !

Le chapitre suivant aborde un point essentiel du projet Jobeet : la sécurisation du backoffice d'administration à l'aide d'un identifiant et d'un mot de passe. Ce sera également l'occasion de parler de la classe Symfony qui gère l'utilisateur courant…

chapitre 13


symfony
Login Required
This page is not public.
How to access this page
You must proceed to the login page and enter your id and password.
What's Next
Proceed to login
Back to previous page

Authentification et droits avec l'objet sfUser

MOTS-CLÉS :

- Authentification
- Droits d'accès
- Objet sfUser

Gérer l'authentification d'un utilisateur, lui accorder des droits d'accès à certaines ressources, ou bien garder en mémoire des informations dans la session en cours est monnaie courante dans une application web actuelle.

Le framework Symfony intègre nativement tous ces mécanismes afin de faciliter la gestion de l'utilisateur qui navigue sur l'application.

Le chapitre précédent a été l'occasion de découvrir tout un lot de nouvelles fonctionnalités propres au framework Symfony. Avec seulement quelques lignes de code PHP, le générateur d'administration de Symfony assure au développeur la création d'interfaces de gestion en quelques minutes.

Les prochaines pages permettent de découvrir comment Symfony gère la persistance des données entre les requêtes HTTP. En effet, le protocole HTTP est dit « sans état », ce qui signifie que chaque requête effectuée est complètement indépendante de celle qui la précède ou de celle qui lui succède. Les sites web d'aujourd'hui nécessitent un moyen de faire persister les données entre les requêtes afin d'améliorer l'expérience de l'utilisateur.

La session d'un utilisateur peut être identifiée à l'aide d'un « cookie ». Dans Symfony, le développeur n'a nul besoin de manipuler directement la session car celle-ci est en fait abstraite grâce à l'objet `sfUser` qui représente l'utilisateur final de l'application.

> CULTURE TECHNIQUE **Qu'est-ce qu'un cookie ?**
>
> Malgré tout ce que l'on a pu lui reprocher dans le passé au sujet de la sécurité, un cookie n'en demeure pas moins un simple fichier texte déposé temporairement sur le poste de l'utilisateur. L'objectif premier du cookie dans une application web est la reconnaissance de l'utilisateur entre deux requêtes HTTP ainsi que le stockage de brèves informations n'excédant pas 4 kilo-octets. Un cookie possède au minimum un nom, une valeur et une date d'expiration dans le temps. D'autres paramètres optionnels supplémentaires peuvent lui être attribués comme son domaine de validité ou bien le fait qu'il soit utilisé sur une connexion sécurisée via le protocole HTTPS. Un cookie est créé par le navigateur du client à la demande du serveur lorsque ce dernier lui envoie les en-têtes adéquats. À chaque nouvelle requête HTTP, si le navigateur du client dispose d'un cookie valable pour le nom de domaine en cours, il transmet le nom et la valeur de son cookie au serveur qui pourra ainsi opérer des traitements particuliers.

Découvrir les fonctionnalités de base de l'objet sfUser

Le chapitre précédent fait quelque peu usage de l'objet `sfUser` qui garde en mémoire les messages de feedback à afficher à l'utilisateur après que celui-ci a réalisé une action. La présente partie explique ce qu'ils sont réellement, comment ils fonctionnent et comment les utiliser dans les développements Symfony.

Comprendre les messages « flash » de feedback

À quoi servent ces messages dans Symfony ?

Dans Symfony, un « flash » est un message éphémère stocké dans la session de l'utilisateur et qui est automatiquement supprimé à la toute prochaine requête. Ces messages sont particulièrement utiles lorsque l'on a besoin d'afficher un message à l'utilisateur après une redirection. Le générateur d'administration a recours à ces messages de feedback dès lors qu'une offre est sauvegardée, supprimée ou bien prolongée dans le temps.

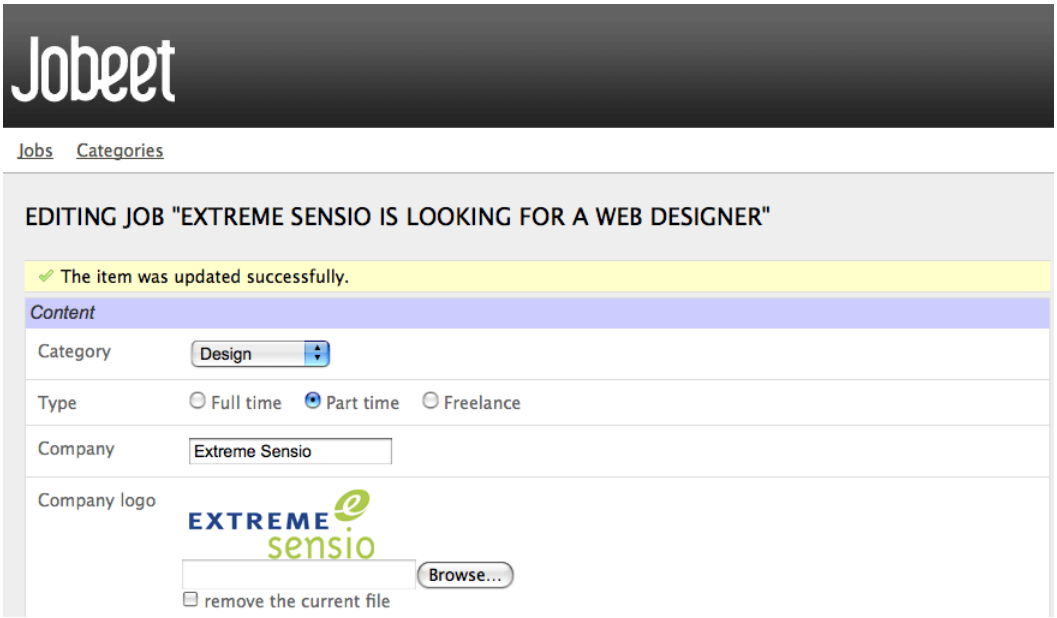


Figure 13–1
Exemple de message flash dans l'administration de Jobeet

Écrire des messages flash depuis une action

Les messages flash de feedback sont généralement définis dans les méthodes des classes d'action après que l'utilisateur a effectué une opération sur l'application. La mise en mémoire d'un message est triviale puisqu'il s'agit simplement d'utiliser la méthode `setFlash()` de l'objet `sfUser` courant comme le montre le code ci-dessous.

Exemple de création d'un message flash dans le fichier apps/frontend/modules/job/actions/actions.class.php

```
public function executeExtend(sfWebRequest $request)
{
  $request->checkCSRFProtection();
```

```
  $job = $this->getRoute()->getObject();
  $this->forward404Unless($job->extend());

  $this->getUser()->setFlash('notice', sprintf('Your job
validity has been extend until %s.', date('m/d/Y',
strtotime($job->getExpiresAt()))));

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

La méthode `setFlash()` accepte deux arguments. Le premier est l'identifiant du message flash tandis que le second est le corps exact du message. Il est possible de définir n'importe quel identifiant de message flash, mais `notice` et `error` sont les plus communs car ils sont principalement utilisés par le générateur d'administration. Ils servent respectivement à afficher des messages d'information et des messages d'erreur.

Lire des messages flash dans un template

C'est au développeur qu'incombe la tâche d'inclure ou non les messages flash dans les templates. Pour ce faire, Symfony intègre les deux méthodes `hasFlash()` et `getFlash()` de l'objet `sfUser`. Ces dernières permettent respectivement de vérifier si l'utilisateur possède ou non un message flash pour l'identifiant passé en paramètre, et de récupérer celui-ci en vue de son affichage dans le template. Dans Jobeet par exemple, les flashs sont tous affichés par le fichier `layout.php`.

Affichage des messages flash dans le fichier 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'objet `sfUser` est accessible via la variable spéciale `$sf_user`.

REMARQUE Accéder à d'autres objets internes de Symfony dans les templates

D'autres objets internes de Symfony sont toujours accessibles dans les templates, sans avoir à les leur passer explicitement depuis une action. Il s'agit par exemple des objets `sfRequest`, `sfUser` ou bien encore `sfResponse` qui se trouvent respectivement dans les variables `$sf_request`, `$sf_user` et `$sf_response`.

Stocker des informations dans la session courante de l'utilisateur

Pour l'instant, les cas d'utilisation de Jobeet ne prévoient aucune contrainte particulière impliquant le stockage d'informations dans la session de l'utilisateur. Pourquoi ne pas ajouter un nouveau besoin fonctionnel

permettant de faire usage de la session ? Il s'agit en effet de développer un mécanisme simple d'historique dans le but de faciliter la navigation de l'utilisateur. À chaque fois que ce dernier consulte une offre, celle-ci est conservée dans l'historique, et les trois dernières offres lues sont réaffichées dans la barre de menu afin de pouvoir y revenir plus tard.

Lire et écrire dans la session de l'utilisateur courant

Pour répondre à cette problématique, il convient de sauvegarder dans la session de l'utilisateur un historique des offres d'emploi lues et d'y ajouter l'offre en cours de consultation à son arrivée. Pour ce faire, le framework Symfony introduit les méthodes `getAttribute()` et `setAttribute()` de l'objet `sfUser` qui permettent respectivement de lire et d'écrire des informations dans la session persistante. Le bout de code ci-dessous illustre le principe de fonctionnement de la session de l'utilisateur.

```
<?php

public function executeAction(sfWebRequest $request)
{
  $this->getUser()->setAttribute('foo', 'bar');// Set bar in the foo session variable

  $foo = $this->getUser()->getAttribute('foo','baz');// Returns bar
}
```

Dans cet exemple, la méthode `setAttribute()` de l'objet `sfUser` sauvegarde la valeur `bar` dans la variable de session `foo`. Puis cette valeur est récupérée au moyen de la méthode `getAttribute()` à laquelle est passé en premier argument le nom de la variable de session. Cette méthode accepte également un second argument facultatif qui correspond à la valeur par défaut à renvoyer si la variable de session est vide ou n'existe pas.

Implémenter l'historique de navigation de l'utilisateur

L'implémentation de l'historique de navigation de l'utilisateur ne pose pas de difficultés particulières. Il s'agit en effet de sauvegarder en session persistante un tableau des clés primaires des offres d'emploi déjà consultées. L'algorithme tient en seulement trois lignes de code PHP comme le montre le code PHP ci-dessous.

Implémentation de l'historique de navigation dans la méthode executeShow() du fichier 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);
  }

  // ...
}
```

Bien qu'il soit possible de stocker des objets PHP directement dans une session, cette pratique n'en demeure pas moins fortement déconseillée dans la mesure où toutes les variables de session sont sérialisées entre chaque requête. Lorsqu'une session est recréée sur une nouvelle page, les objets qu'elle contient sont désérialisés, c'est-à-dire qu'ils sont reconstruits et qu'ils peuvent être altérés ou bien obsolètes s'ils ont été modifiés ou bien supprimés entre-temps.

Refactoriser le code de l'historique de navigation dans le modèle

L'action `executeShow()` n'est pas l'endroit le plus idéal pour accueillir la logique de l'historique de navigation. En effet, le code en devient entièrement dépendant, ce qui l'empêche d'être réutilisé ailleurs. D'autre part, il n'est pas factorisé et complexifie ainsi inutilement l'action `executeShow()`.

Implémenter l'historique de navigation dans la classe myUser

D'après ce dernier postulat, on en déduit rapidement le besoin de déplacer le code vers la couche du modèle afin de respecter la séparation entre les différentes logiques. Le meilleur endroit pour recevoir l'implémentation de l'historique de navigation est naturellement la classe `myUser` qui surcharge les spécificités de la classe de base `sfUser` avec des comportements propres à l'application courante.

Implémentation de la méthode addJobToHistory() dans la classe myUser du fichier 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 remaniement du code prend désormais en compte toutes les contraintes de l'historique de navigation. L'instruction `in_array($job->getId(), $ids)` s'assure que l'offre d'emploi n'existe pas deux fois dans l'historique de navigation tandis que l'instruction `array_slice($ids, 0, 3)` se contente de récupérer uniquement les clés primaires des trois dernières annonces consultées par l'utilisateur en vue de réafficher un lien vers chacune d'elle dans la page courante.

Simplifier l'action executeShow() de la couche contrôleur

L'étape suivante consiste à mettre à jour l'action `executeShow()` précédente afin de la simplifier en lui implémentant la méthode `addJobToHistory()`.

Remaniement de la méthode executeShow() dans le fichier 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);
  }

  // ...
}
```

Grâce au remaniement du code, l'action ne comporte plus que deux lignes de code PHP alors qu'il en fallait quatre précédemment. La dernière étape consiste enfin à récupérer les offres d'emploi de l'historique puis afficher leur titre respectif sur la page en cours.

Afficher l'historique des offres d'emploi consultées

À présent, il ne reste plus qu'à ajouter le programme PHP qui génère dans la vue le code HTML de l'historique des trois dernières annonces. L'implémentation de ce script est à ajouter avant la variable `$sf_content` du fichier `layout.php` comme l'illustre le code ci-dessous.

Génération de l'historique des offres dans le fichier 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>
```

Ce template fait appel à la méthode `getJobHistory()` de l'objet `sfUser`. Cette méthode a pour rôle de récupérer les trois derniers objets `JobeetJob` de la base de données à partir des identifiants des offres sauvegardés dans la session courante de l'utilisateur. Le fonctionnement de la méthode `getJobHistory()` est trivial puisqu'il s'agit de faire appel à l'historique des offres dans la session, puis de retourner l'objet `Doctrine_Collection` résultant de la requête SQL exécutée par Doctrine.

Implémentation de la méthode getJobHistory() dans la classe myUser du fichier apps/frontend/lib/myUser.class.php

```
class myUser extends sfBasicSecurityUser
{
  public function getJobHistory()
  {
    $ids = $this->getAttribute('job_history', array());

    if (!empty($ids))
    {
      return Doctrine::getTable('JobeetJob')
        ->createQuery('a')
        ->whereIn('a.id', $ids)
        ->execute();
    }
    else
    {
      return array();
    }
  }

  // ...
}
```

La capture d'écran ci-dessous présente le résultat final obtenu après que l'historique de navigation a été complètement implémenté.

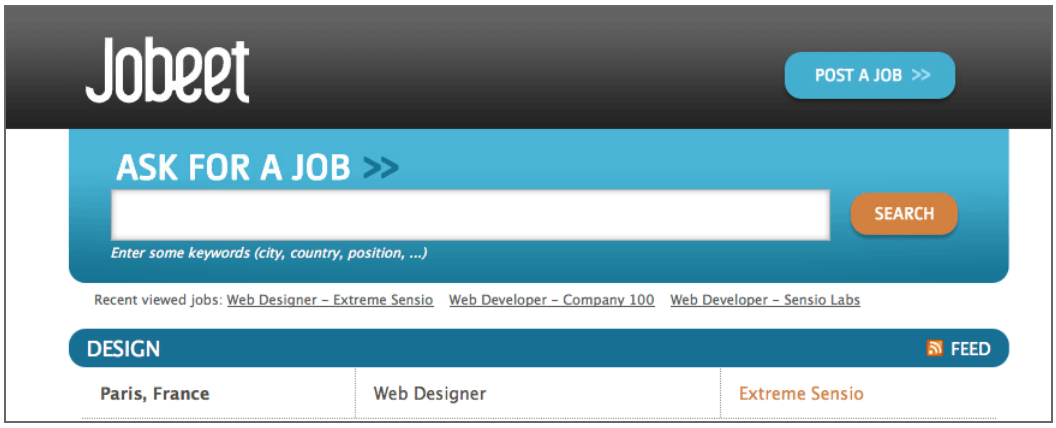


Figure 13–2
Exemple de l'historique de navigation reposant sur les informations sauvegardées en session

Implémenter un moyen de réinitialiser l'historique des offres consultées

Tous les attributs de l'utilisateur sont gérés par une instance de la classe `sfParameterHolder`. Les méthodes `getAttribute()` et `setAttribute()` sont deux méthodes « proxy » (raccourcies) pour `getParameterHolder->get()` et `getParameterHolder()->set()`.

L'objet `sfParameterHolder` contient également une méthode `remove()` qui permet de supprimer un attribut de la session de l'utilisateur. Cette méthode n'est associée à aucune méthode raccourcie dans la classe `sfUser`, c'est pourquoi le code ci-dessous fait directement appel à l'objet `sfParameterHolder` pour supprimer l'attribut `job_history`, et ainsi vider l'historique des offres consultées.

Implémentation de la méthode resetJobHistory dans la classe myUser du fichier apps/frontend/lib/myUser.class.php

```
class myUser extends sfBasicSecurityUser
{
  public function resetJobHistory()
  {
    $this->getAttributeHolder()->remove('job_history');
  }

  // ...
}
```

La classe `sfParameterHolder` est également utilisée par l'objet `sfRequest` pour sauvegarder ses différents paramètres.

Les sections suivantes de ce chapitre abordent de nouvelles notions clés du framework Symfony. Il s'agit en effet de découvrir les mécanismes internes de sécurisation des applications et de contrôle de droits d'accès de l'utilisateur. Un système d'authentification avec nom d'utilisateur et mot de passe pour l'application backend de Jobeet est développé en guise d'exemple pratique.

Comprendre les mécanismes de sécurisation des applications

Cette partie s'intéresse aux principes d'authentification et de contrôle de droits d'accès sur une application. L'authentification consiste à s'assurer que l'utilisateur courant est bien authentifié sur l'application ; c'est par exemple le cas lorsqu'il remplit un formulaire avec son couple d'identifiant et mot de passe valides. Le contrôle d'accès, quant à lui, vérifie que l'utilisateur dispose des autorisations nécessaires pour accéder à tout ou partie d'une application (par exemple, lorsqu'il s'agit de lui empêcher la suppression d'un objet s'il ne dispose pas d'un statut de super administrateur).

Activer l'authentification de l'utilisateur sur une application

Découvrir le fichier de configuration security.yml

Comme avec la plupart des fonctionnalités de Symfony, la sécurité d'une application est gérée au travers du fichier YAML `security.yml`. Pour l'instant, ce fichier se trouve par défaut dans le répertoire `apps/backend/config/` du projet et désactive toute sécurité de l'application comme le montre son contenu :

Contenu du fichier apps/backend/config/security.yml

```
default:
  is_secure: off
```

En fixant la constante de configuration `is_secure` à la valeur `on`, toute l'application backend forcera l'utilisateur à être authentifié pour aller plus loin. La capture d'écran ci-dessous illustre la page affichée par défaut à l'utilisateur si ce dernier n'est pas authentifié.

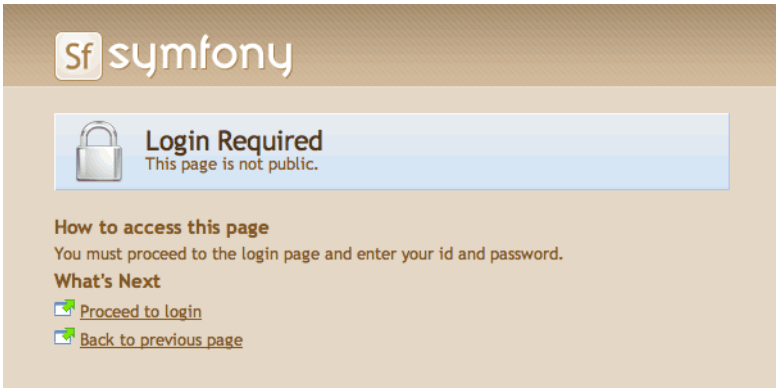


Figure 13–3
Page de login par défaut de Symfony pour un utilisateur non identifié

Dans un fichier de configuration YAML, une valeur booléenne peut être exprimée à l'aide des chaînes de caractères `on`, `off`, `true` ou bien `false`. Comment se fait-il que l'utilisateur se voit redirigé vers cette page qui n'apparaît nulle part dans le projet ? La réponse se trouve tout naturellement dans les logs que Symfony génère en environnement de développement.

Analyse des logs générés par Symfony

L'analyse des logs dans la barre de débogage de Symfony indique que la méthode `executeLogin()` de la classe `defaultActions` est appelée pour chaque page à laquelle l'utilisateur tente d'accéder.

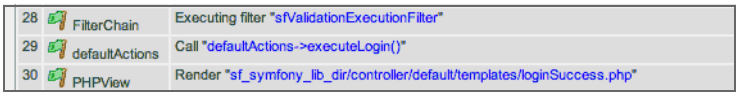


Figure 13–4
Extrait des logs lorsque l'utilisateur tente d'accéder à une page sécurisée

Le module `default` n'existe pas réellement dans un projet Symfony puisqu'il s'agit d'un module livré entièrement avec le framework. Bien évidemment, l'appel à la méthode `executeLogin()` du module `default` est entièrement redéfinissable afin de pouvoir rediriger automatiquement l'utilisateur vers une page personnalisée.

Personnaliser la page de login par défaut

Lorsqu'un utilisateur essaie d'accéder à une action sécurisée, Symfony délègue automatiquement la requête à l'action `login` du module `default`. Ces deux informations ne sont pas choisies au hasard par le framework puisqu'elles figurent dans le fichier de configuration `settings.yml` de l'application. Ainsi, il est possible de redéfinir soi-même l'action personnalisée à invoquer lorsqu'un utilisateur souhaite accéder à une page sécurisée.

Redéfinition de l'action login dans le fichier apps/backend/config/settings.yml

```
all:
  .actions:
    login_module: default
    login_action: login
```

Pour des raisons techniques évidentes, il est impossible de sécuriser la page de login afin d'éviter une récursivité infinie. Au chapitre 4, il a été démontré qu'une même configuration s'étalonne à plusieurs endroits dans le projet. Il en va de même pour le fichier `security.yml`. En effet, pour sécuriser ou retirer la sécurité d'une action ou de tout un module de l'application, il suffit d'ajouter un fichier `security.yml` dans le répertoire `config/` du module concerné.

```
index:
  is_secure: off

all:
  is_secure: on
```

Authentifier et tester le statut de l'utilisateur

Par défaut, la classe `myUser` dérive directement de la classe `sfBasicSecurityUser` qui étend elle-même la classe `sfUser`. `sfBasicSecurityUser` intègre des méthodes additionnelles pour gérer l'authentification et les droits d'accès de l'utilisateur courant. L'authentification de l'utilisateur se manipule avec deux méthodes seulement : `isAuthenticated()` et `setAuthenticated()`. La première se contente de retourner si oui ou non l'utilisateur est déjà authentifié, alors que la seconde permet de l'authentifier (ou de le déconnecter). Le bout de code illustre leur fonctionnement dans le cadre d'une action.

```
if (!$this->getUser()->isAuthenticated())
{
  $this->getUser()->setAuthenticated(true);
}
```

À présent, il est temps de s'intéresser au mécanisme natif de contrôle de droits d'accès. La section suivante explique comment restreindre tout ou partie des fonctionnalités d'une application à l'utilisateur en lui affectant un certain nombre de droits.

Restreindre les actions d'une application à l'utilisateur

Dans la plupart des projets complexes, les développeurs sont confrontés à la notion de politique de droits d'accès aux informations. C'est d'autant

plus vrai dans le cas d'une interface de gestion ou bien dans un Intranet pour lequel il peut exister plusieurs profils d'utilisateurs : administrateur, publicateur, modérateur, trésorier… Tous ne possèdent pas les mêmes autorisations et ne peuvent dans ce cas avoir accès à certaines parties de l'application lorsqu'ils sont connectés. Heureusement, Symfony intègre parfaitement un système de contrôle de droits d'accès simple et rapide à mettre en œuvre.

Activer le contrôle des droits d'accès sur l'application

Lorsqu'un utilisateur est authentifié sur l'application, il ne doit pas forcément avoir accès à toutes les fonctionnalités de cette dernière. Certaines zones peuvent donc lui être restreintes en établissant une politique de droits d'accès. L'usager doit alors posséder les droits nécessaires et suffisants pour atteindre les pages qu'il désire. Dans Symfony, les droits de l'utilisateur sont nommés **credentials** et se déclarent à plusieurs niveaux. Le code ci-dessous du fichier `security.yml` de l'application désactive l'authentification mais contraint néanmoins l'utilisateur à posséder les droits d'administrateur pour aller plus loin.

```
default:
  is_secure:   off
  credentials: admin
```

Le système de contrôle de droits d'accès de Symfony est particulièrement simple et puissant. Un droit peut représenter n'importe quelle chose pour décrire le modèle de sécurité de l'application comme les groupes ou les permissions.

Établir des règles de droits d'accès complexes

La section `credentials` du fichier `security.yml` supporte les opérations booléennes pour décrire les contraintes de droits d'accès complexes. Par exemple, si un utilisateur est contraint d'avoir les droits A et B, il suffit d'encadrer ces deux derniers avec des crochets.

```
index:
  credentials: [A, B]
```

En revanche, si un utilisateur doit avoir les droits A ou B, il faut alors encadrer ces derniers par une double paire de crochets comme ci-dessous.

```
index:
  credentials: [[A, B]]
```

Au final, il est possible de mixer à volonté les règles booléennes jusqu'à trouver celle qui correspond à la politique de droits d'accès que l'on souhaite mettre en application.

Gérer les droits d'accès via l'objet sfBasicSecurityUser

Toute la politique de droits d'accès peut également être gérée directement grâce à l'objet `sfBasicSecurityUser` qui fournit un ensemble de méthodes capables d'ajouter ou de retirer des droits à l'utilisateur, mais qui permet également de tester si ce dernier en possède certains, comme le montrent les exemples ci-dessous.

```
// 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
```

Le tableau ci-dessous résume et détaille plus exactement chacune de ces méthodes.

Tableau 13–1 Liste des méthodes de l'objet sfBasicSecurityUser

Nom de la méthode	Description
`addCredential('foo')`	Affecte un droit à l'utilisateur
`addCredentials('foo', 'bar')`	Affecte un ou plusieurs droits à l'utilisateur
`hasCredential('foo')`	Indique si oui ou non l'utilisateur possède le droit `foo`
`hasCredential(array('foo', 'bar'))`	Indique si oui ou non l'utilisateur possède les droits `foo` et `bar`
`hasCredential(array('foo', 'bar'), false)`	Indique si oui ou non l'utilisateur possède l'un des deux droits
`removeCredential('foo')`	Retire le droit `foo` à l'utilisateur
`clearCredentials()`	Retire tous les droits de l'utilisateur

Pour l'application Jobeet, il n'est pas nécessaire de gérer les droits d'accès dans la mesure où celle-ci n'accueille qu'un seul type de profils : le rôle administrateur.

Mise en place de la sécurité de l'application backend de Jobeet

Tous les concepts présentés dans la section précédente sont plutôt théoriques et n'ont pas encore été véritablement mis en application. Il est temps de retourner à l'application backend et de lui ajouter la page d'identification qui lui manque pour le moment. L'objectif n'est pas d'écrire ce type de fonctionnalité ex nihilo (*from scratch* disent les puristes anglicistes), et heureusement le framework Symfony dispose de tout le nécessaire pour mettre cela en œuvre en quelques minutes. Il s'agit en effet de recourir à l'installation du plugin `sfDoctrineGuardPlugin` qui intègre entre autres les mécanismes d'identification et de reconnaissance de l'utilisateur.

Installation du plug-in sfDoctrineGuardPlugin

L'une des incroyables forces du framework Symfony réside dans son riche écosystème de plugins. L'un des prochains chapitres de cet ouvrage explique en quoi il est très facile de créer des plugins et en quoi ces derniers sont des outils puissants et pratiques. En effet, un plugin est capable de contenir aussi bien des modules que de la configuration, des classes PHP, des fichiers XML ou encore des ressources web… Pour le développement de l'application Jobeet, c'est le plugin `sfDoctrineGuardPlugin` qui sera installé pour garantir le besoin de sécurisation de l'interface d'administration.

L'installation d'un plugin dans le projet est simple puisqu'il suffit d'exécuter une commande depuis l'interface en ligne de commande Symfony comme le montre le code suivant.

```
$ php symfony plugin:install sfDoctrineGuardPlugin
```

La commande `plugin:install` télécharge et installe un plugin à partir de son nom. Tous les plugins du projet sont stockés sous le répertoire `plugins/` et chacun d'eux possède son propre répertoire nommé avec le nom du plugin. Bien qu'elle soit pratique et souple à utiliser, la tâche `plugin:install` requiert l'installation de PEAR sur le serveur pour fonctionner correctement !

Lorsque l'on installe un plugin à partir de la tâche `plugin:install`, Symfony télécharge la toute dernière version stable de ce dernier. Pour installer une version spécifique d'un plugin, il suffit de lui passer l'option facultative `--release` accompagnée du numéro de la version désirée. La page dédiée du plugin sur le site officiel du framework Symfony liste toutes les versions disponibles du plugin pour chaque version du framework.

Dans la mesure où un plugin est copié en intégralité dans son propre répertoire, il est également possible de télécharger son archive depuis le site officiel de Symfony, puis de la décompresser dans le répertoire `plugins/` du projet. Enfin, une dernière méthode alternative d'installation consiste à créer un lien externe vers le dépôt Subversion du plugin à l'aide d'un client Subversion et de la propriété `svn:externals` sur le répertoire `plugins/`.

Enfin, il ne faut pas oublier d'activer le plugin après l'avoir installé si l'on n'utilise pas la méthode `enableAllPluginsExcept()` de la classe `config/ProjectConfiguration.class.php`.

Mise en place des sécurités de l'application backend

Générer les classes de modèle et les tables SQL

Chaque plugin possède son propre fichier `README` qui explique comment l'installer et le configurer pour le projet. Les lignes qui suivent décrivent pas à pas la configuration du plugin `sfDoctrineGuardPlugin` en commençant par la génération du modèle et des nouvelles tables SQL. En effet, ce plugin fournit plusieurs classes de modèle pour gérer les utilisateurs, les groupes et les permissions sauvegardés en base de données.

```
$ php symfony doctrine:build-all-reload
```

La tâche `doctrine:build-all-reload` supprime toutes les tables existantes de la base de données avant de les recréer une par une. Afin d'éviter cela, il est possible de générer le modèle, les formulaires et les filtres, et enfin créer les nouvelles tables en exécutant le script SQL du plugin généré dans le répertoire `data/sql/`.

Implémenter de nouvelles méthodes à l'objet User via la classe sfGuardSecurityUser

L'exécution de la tâche `doctrine:build-all-reload` a généré de nouvelles classes de modèle pour le plugin `sfDoctrineGuardPlugin`, c'est pourquoi le cache du projet doit être vidé pour les prendre en compte.

```
$ php symfony cc
```

Dans la mesure où `sfDoctrineGuardPlugin` ajoute plusieurs nouvelles méthodes à la classe de l'utilisateur, il est nécessaire de changer la classe parente de la classe `myUser` par `sfGuardSecurityUser` comme le montre le code ci-dessous.

Redéfinition de la classe de base de myUser dans le fichier apps/backend/lib/myUser.class.php

```
class myUser extends sfGuardSecurityUser
{
  // ...
}
```

Activer le module sfGuardAuth et changer l'action de login par défaut

Le plugin `sfDoctrineGuardPlugin` fournit également une action `signin` à l'intérieur du module `sfGuardAuth` afin de gérer l'authentification des utilisateurs. Il faut donc indiquer à Symfony que c'est vers cette action que les utilisateurs non authentifiés doivent être amenés lorsqu'ils essaient d'accéder à une page sécurisée. Pour ce faire, il suffit d'éditer le fichier de configuration `settings.yml` de l'application backend.

Définition du module et de l'action par défaut pour la page de login dans le fichier apps/backend/config/settings.yml

```
all:
  .settings:
    enabled_modules: [default, sfGuardAuth]

    # ...

  .actions:
    login_module:    sfGuardAuth
    login_action:    signin

    # ...
```

Dans la mesure où tous les plugins sont partagés pour toutes les applications du projet, il est nécessaire de n'activer que les modules à utiliser dans l'application en les ajoutant explicitement au paramètre de configuration `enabled_modules` comme c'est le cas ici pour le module `sfGuardAuth`. La figure ci-dessous illustre la page de login qui est affichée à l'utilisateur lorsque ce dernier n'est pas authentifié sur l'application.

Figure 13–5
Page d'identification à l'application backend

Créer un utilisateur administrateur

La dernière étape consiste à enregistrer dans la base de données un compte utilisateur autorisé à s'authentifier sur l'interface de gestion de Jobeet. Il serait bien sûr possible de réaliser cette opération manuellement directement dans la base de données ou bien à partir d'un fichier de données initiales mais le plugin fournit des tâches Symfony pour faciliter ce genre de procédures.

```
$ php symfony guard:create-user fabien SecretPass
$ php symfony guard:promote fabien
```

La tâche `guard:create-user` permet de créer un nouveau compte utilisateur dans la base de données en lui spécifiant le nom d'utilisateur en premier argument et le mot de passe associé en second. De son côté, la tâche `guard:promote` promeut le compte utilisateur passé en argument comme super administrateur.

`sfDoctrineGuardPlugin` inclut d'autres tâches pour gérer les utilisateurs, les groupes et les permissions depuis la ligne de commande. Par exemple, l'utilisation de la tâche `list` affiche la liste des commandes disponibles sous l'espace de nom `guard`.

```
$ php symfony list guard
```

Cacher le menu de navigation lorsque l'utilisateur n'est pas authentifié

Il reste encore un petit détail à régler. En effet, les liens du menu d'administration de l'interface backend continuent d'être affichés même quand l'utilisateur n'est pas authentifié. Ce dernier ne devrait donc pas être en mesure de voir ce menu. Pour le masquer, il suffit seulement de tester dans le template si l'utilisateur est authentifié ou non grâce à la méthode `isAuthenticated()` vue précédemment.

Masquer le menu de navigation à l'utilisateur non identifié dans le fichier 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>
      <li><?php echo link_to('Logout', '@sf_guard_signout') ?></li>
    </ul>
  </div>
<?php endif; ?>
```

Un lien de déconnexion a également été ajouté pour permettre à l'utilisateur connecté de fermer proprement sa session sur l'interface d'administration. Ce lien utilise la route `sf_guard_signout` déclarée dans le plugin `sfDoctrineGuardPlugin`. La tâche `app:routes` permet de lister l'ensemble des routes définies pour l'application courante.

Ajouter un nouveau module de gestion des utilisateurs

La fin de ce chapitre est toute proche mais il est encore temps d'ajouter un module complet de gestion des utilisateurs pour parfaire l'application. Cette opération ne demande que quelques secondes puisque `sfDoctrineGuardPlugin` détient ce précieux module. De la même manière que pour le module `sfGuardAuth`, le plugin `sfGuardUser` doit être référencé auprès de la liste des modules activés dans le fichier de configuration `settings.yml`.

Ajout du module sfGuardUser dans le fichier apps/backend/config/settings.yml

```
all:
  .settings:
    enabled_modules: [default, sfGuardAuth, sfGuardUser]
```

Le module `sfGuardUser` a été généré à partir du générateur d'administration étudié au chapitre précédent. De ce fait, il est entièrement paramétrable et personnalisable grâce au fichier de configuration `generator.yml` se trouvant dans le répertoire `config/` du module.

Il ne reste finalement plus qu'à installer un lien dans le menu de navigation afin de permettre à l'administrateur d'accéder à ce nouveau module pour gérer tous les comptes utilisateurs de l'application.

```
<?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>
```

```
      <li><?php echo link_to('Users', '@sf_guard_user') ?></li>
      <li><?php echo link_to('Logout', '@sf_guard_signout') ?></li>
    </ul>
  </div>
<?php endif; ?>
```

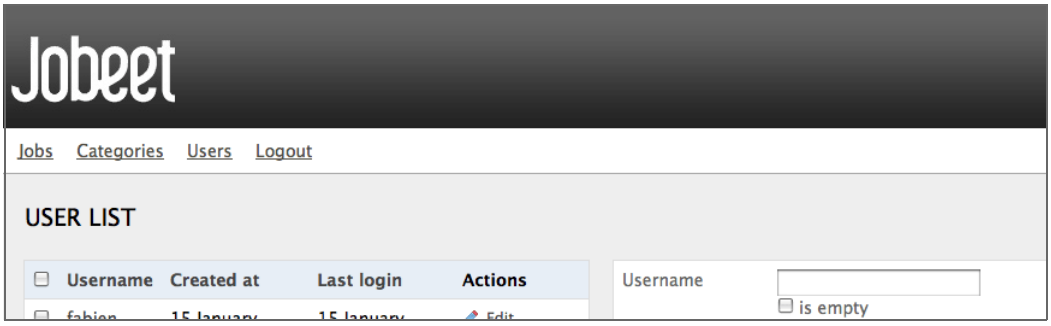


Figure 13–6
Rendu final de la page d'accueil du module sfGuardUser

La capture d'écran ci-dessus illustre le rendu final du menu de navigation et du gestionnaire d'utilisateurs qui ont été ajoutés à l'application en quelques minutes seulement.

Implémenter de nouveaux tests fonctionnels pour l'application frontend

Ce chapitre n'est pas encore terminé puisqu'il reste à parler rapidement des tests fonctionnels propres à l'utilisateur. Comme le navigateur de Symfony est capable de simuler les cookies, il est très facile de tester les comportements de l'usager à l'aide du testeur natif `sfTesterUser`. Il est temps de mettre à jour les tests fonctionnels de l'application frontend pour prendre en compte les fonctionnalités additionnelles du menu implémentées dans ce chapitre. Pour ce faire, il suffit d'ajouter le code suivant à la fin du fichier de tests fonctionnels `jobActionsTests.php`.

Tests fonctionnels de l'historique de navigation à ajouter à la fin du fichier 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()
;
```

Afin de faciliter les tests, il est nécessaire de forcer le rechargement des données initiales de test et de réinitialiser le navigateur afin de démarrer avec une nouvelle session vierge. Les tests ci-dessus font appel à la méthode `isAttribute()` du testeur `sfTesterUser` qui permet de vérifier la présence et la valeur d'une donnée de session de l'utilisateur.

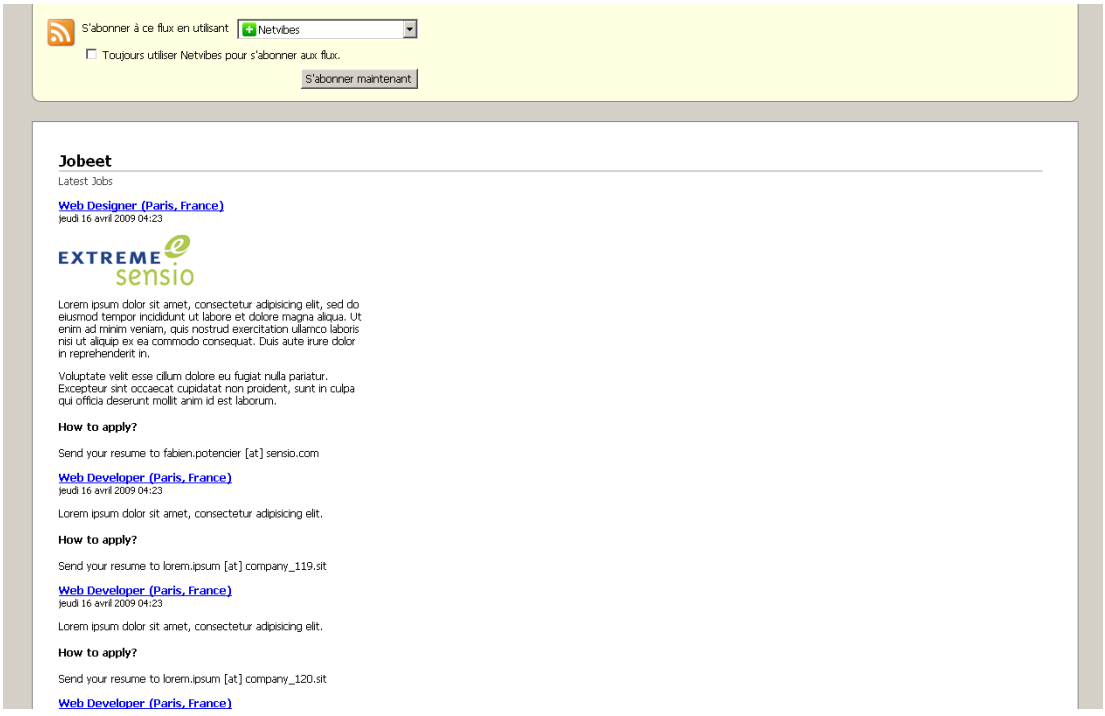
Le testeur `sfTesterUser` fournit également les méthodes `isAuthenticated()` et `hasCredential()` qui contrôlent l'authentification et les autorisations de l'utilisateur courant.

En résumé…

Les classes internes de Symfony dédiées à l'utilisateur constituent une bonne manière de s'abstraire de la gestion du mécanisme des sessions de PHP. Couplées à l'excellent système de gestion des plugins ainsi qu'au plugin `sfDoctrineGuardPlugin`, elles sont capables de sécuriser une interface d'administration en quelques minutes ! Au final, l'application Jobeet dispose d'une interface de gestion propre et complète qui permet aux administrateurs de gérer des utilisateurs grâce aux modules livrés par le plugin.

Comme Jobeet est une application Web 2.0 digne de ce nom, elle ne peut échapper aux traditionnels flux RSS et Atom qui seront développés avec autant de facilité au cours du prochain chapitre…

chapitre 14


S'abonner à ce flux en utilisant
Netvibes
Toujours utiliser Netvibes pour s'abonner aux flux.
S'abonner maintenant
Jobeet
Latest Jobs
Web Designer (Paris, France)
jeudi 16 avril 2009 04:23
EXTREME sensio
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
Web Developer (Paris, France)
jeudi 16 avril 2009 04:23
Lorem ipsum dolor sit amet, consectetur adipisicing elit.
How to apply?
Send your resume to lorem.ipsum [at] company_119.sit
Web Developer (Paris, France)
jeudi 16 avril 2009 04:23
Lorem ipsum dolor sit amet, consectetur adipisicing elit.
How to apply?
Send your resume to lorem.ipsum [at] company_120.sit
Web Developer (Paris, France)

Les flux de syndication ATOM

Toutes les applications web modernes mettent à disposition leurs contenus sous forme de flux afin de permettre aux internautes de suivre les dernières informations publiées dans leur navigateur ou leur agrégateur favoris. L'application Jobeet ne déroge pas à cette règle, et, grâce au framework Symfony, sera pourvue d'un système de flux de syndication ATOM.

MOTS-CLÉS :

- Formats XML, HTML et ATOM
- Routage
- Templates

La fraîcheur et le renouvellement de l'information sont des points majeurs qui contribuent à la réussite d'une application web grand public. En effet, un site Internet dont le contenu n'est pas mis à jour régulièrement risque de perdre une part non négligeable de son audience, cette dernière ayant le besoin permanent d'être nourrie d'informations nouvelles.

Or, comment est-il possible d'informer un internaute non connecté au site que le contenu de ce dernier a été mis à jour ? Par exemple, en ce qui concerne l'application développée tout au long de cet ouvrage, il s'agit de trouver un moyen de notifier à l'internaute la présence de nouvelles offres d'emploi publiées, sans que celui-ci n'ait à se rendre de lui-même sur le site. La réponse à cette problématique se trouve dans les flux de syndication (*feeds* en anglais) RSS et ATOM. En effet, les formats RSS et ATOM sont deux standards reposant sur la norme XML, et peuvent être lus par tous les navigateurs web modernes ou par des agrégateurs de contenus tels que Netvibes, Google Reader, delicious.com… Leur standardisation ainsi que leur extrême simplicité servent également à échanger, voire à publier, de l'information entre les différentes applications web ou terminaux (téléphones mobiles par exemple).

L'objectif de ce quatorzième chapitre est d'implémenter petit à petit des flux de syndication ATOM des offres d'emploi afin que l'utilisateur puisse être tenu informé des nouveautés publiées.

Découvrir le support natif des formats de sortie

Définir le format de sortie d'une page

Le framework Symfony dispose d'un support natif des formats de sortie et des types de fichiers mimes (*mime-types* en anglais), ce qui signifie que le même Modèle et Contrôleur peuvent avoir différents templates en fonction du format demandé. Le format par défaut est bien évidemment le HTML mais Symfony supporte un certain nombre de formats de sortie supplémentaires comme `txt`, `js`, `css`, `json`, `xml`, `rdf` ou bien encore `atom`.

La méthode `setRequestFormat()` de l'objet `sfRequest` permet de définir le format de sortie d'une page.

```
$request->setRequestFormat('xml');
```

Gérer les formats de sortie au niveau du routage

Bien qu'il soit possible de définir manuellement le format de sortie d'une action dans Symfony, ce dernier se trouve embarqué la plupart du temps dans l'URL. De ce fait, Symfony est capable de déterminer lui-même le format de sortie à retourner d'après la valeur de la variable `sf_format` de la route correspondante. Par exemple, pour la liste des offres d'emploi, l'url est la suivante :

http://jobeet.localhost/frontend_dev.php/job

Cette même URL est équivalente à la suivante :

http://jobeet.localhost/frontend_dev.php/job.html

Ces deux URLs sont effectivement identiques car les routes générées par la classe `sfDoctrineRouteCollection` possèdent la variable `sf_format` en guise d'extension, et parce que le HTML est le format privilégié. Pour s'en convaincre, il suffit d'exécuter la commande `app:routes` afin d'obtenir un résultat similaire à la capture ci-dessous.

```
~/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 $
```

Figure 14–1
Liste des routes paramétrées pour l'application frontend

Présentation générale du format ATOM

Un fichier de syndication ATOM est en réalité un document au format XML qui s'appuie sur une structure bien définie, localisée dans sa déclaration de type de document : la DTD (*Document Type Declaration* en anglais). L'ensemble des spécificités du format ATOM dépasse largement le cadre de cet ouvrage ; il est néanmoins nécessaire de connaître les fondamentaux pour être capable de réaliser un flux minimal valide. La principale chose à retenir à propos du format ATOM concerne sa structure. En effet, un flux ATOM est composé de deux parties distinctes : les informations générales du flux et les entrées.

Les informations globales du flux

Les informations générales sont situées tout de suite sous l'élément racine `<feed>` du flux. Les balises présentes sous cet élément apportent des données globales comme le titre du flux, sa description, son lien, sa ou ses catégories, sa date de dernière mise à jour, son logo… Certaines d'entre elles sont obligatoires et doivent donc figurer impérativement dans le flux afin que celui-ci soit considéré comme valide.

Les entrées du flux

Les entrées, quant à elles, sont les items qui décrivent le contenu. Leur nombre n'est pas limité et elles sont référencées à l'aide de l'élément `<entry>` qui contient une série de nœuds fils pour les décrire et donner du sens à l'information syndiquée. Les fils du nœud `<entry>` sont ainsi responsables d'informations telles que le titre, le contenu, le lien vers la page originale sur le site Internet, le ou les auteurs, la date de publication… et bien plus encore. Là encore, certaines données sont obligatoires afin de rendre le flux valide.

Le flux ATOM minimal valide

Le code ci-dessous donne la structure minimale requise pour rendre un flux ATOM valide. Il intègre entre autres le titre, la date de mise à jour ainsi que l'identifiant unique en guise d'informations générales. En ce qui concerne les entrées, cet exemple est composé d'une seule et unique entrée qui contient elle aussi un jeu d'informations obligatoires. Parmi elles se trouvent le titre du contenu, son extrait, sa date de mise à jour ainsi que son auteur.

Exemple de flux ATOM minimaliste valide

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">

<title>Jobeet</title>
<updated>2009-03-17T20:38:43Z</updated>
<id>afba34a2a11ab13eeba5d0a7aa22bbb6120e177b</id>

  <entry>
   <title>Sensio Labs is looking for a Web Developer</title>
   <author>
    <name>Sensio Labs</name>
   </author>
   <id>d0be2dc421be4fcd0172e5afceea3970e2f3d940</id>
   <updated>2009-03-17T20:38:43Z</updated>
```

```
   <summary>
    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.
   </summary>
</entry>

</feed>
```

L'objectif des prochaines pages est de s'appuyer sur ces connaissances de base dans le but de générer des flux d'informations plus complexes et valides. Il s'agit en effet de construire successivement deux flux ATOM pour l'application Jobeet. Le premier consiste à créer la liste des dernières offres d'emploi publiées sur le site, toutes catégories confondues, alors que le second est un flux dynamique propre à chaque catégorie de l'application.

Générer des flux de syndication ATOM

Afin de s'initier et de comprendre plus concrètement comment fonctionne le mécanisme des formats de sortie dans Symfony, les pages suivantes déroulent pas à pas la création de flux de syndication au format ATOM. Pour commencer, il est primordial de découvrir et de comprendre de quelle manière est déclaré un nouveau format de sortie dans Symfony.

Flux ATOM des dernières offres d'emploi

Déclarer un nouveau format de sortie

Dans Symfony, supporter différents formats est aussi simple que de créer différents templates dont le nom du fichier intègre la particule du format souhaité. Par exemple, pour réaliser un flux de syndication ATOM des dernières offres d'emploi, un nouveau template nommé `indexSuccess.atom.php` doit être disponible et contenir par exemple le contenu statique suivant.

Exemple de code ATOM pour les dernières offres dans le fichier 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>
```

Le nom du fichier contient la particule `atom` avant l'extension `.php`. Cette dernière indique à Symfony le format de sortie à renvoyer au client. La section suivante donne un rapide rappel sur les conventions de nommage des fichiers de template dans un projet.

Rappel des conventions de nommage des templates

Dans la mesure où le format HTML est le plus couramment employé dans la réalisation d'applications web, l'expression du format de sortie `.html` n'est pas obligatoire et peut donc être omise du nom du gabarit PHP. En effet, les deux templates `indexSuccess.php` et `indexSuccess.html.php` sont équivalents pour le framework, c'est pourquoi celui-ci utilise le premier qu'il trouve.

Pourquoi les noms des templates par défaut sont-ils suffixés avec `Success` ? Une action est capable de retourner une valeur qui indique quel template doit être rendu. Si l'action ne retourne rien, cela correspond au code ci-dessous qui renvoie la valeur `Success` :

```
return sfView::SUCCESS; // == 'Success'
```

Pour changer le suffixe d'un template, il suffit tout simplement de retourner une valeur différente comme par exemple :

```
return sfView::ERROR; // == 'Error'
return 'Foo';
```

De même, il a été montré au cours des chapitres précédents que le nom du template à rendre pouvait lui aussi être modifié grâce à l'emploi de la méthode `setTemplate()`.

```
$this->setTemplate('foo');
```

Il est temps de revenir à la génération des flux de syndication et de modifier le layout de l'application grand public afin qu'elle dispose des liens vers ces derniers.

Ajouter le lien vers le flux des offres dans le layout

Par défaut, Symfony modifie automatiquement l'en-tête HTTP `Content-Type` de la réponse en fonction du format. De plus, tous les formats qui ne sont pas du HTML ne sont pas décorés par le layout. Dans le cas des flux ATOM par exemple, Symfony retourne au client le type de contenu `application/atom+xml; charset=utf-8` dans l'en-tête `Content-Type` de la réponse.

L'application frontend de Jobeet a besoin d'un hyperlien supplémentaire pour faciliter l'accès au flux d'informations par l'utilisateur. Le code ci-dessous donne le code HTML et PHP à ajouter dans le pied de page du layout de l'application.

Ajout d'un lien vers le flux de syndication ATOM des offres d'emploi dans le fichier apps/frontend/templates/layout.php

```
<li class="feed">
  <a href="<?php echo url_for('@job?sf_format=atom') ?>">Full
feed</a>
</li>
```

L'URL interne ici créée est la même que celle qui existe déjà pour la liste des offres, à ceci près qu'elle redéfinit la valeur de la variable `sf_format` vue précédemment. Par ailleurs, les navigateurs web sont capables de découvrir et de charger automatiquement les flux de syndication d'une application web, à condition que cette dernière intègre une balise `<link>` dans la section `<head>` de la page. Cette balise spéciale informe le client qu'une ressource externe à la page courante est disponible à l'URL indiquée, et que le type de contenu de cette dernière est du même type que celui spécifié dans l'attribut `type` de la balise.

Ajout du marqueur du flux ATOM dans la section HEAD du fichier apps/frontend/templates/layout.php

```
<link rel="alternate" type="application/atom+xml" title="Latest
Jobs"
  href="<?php echo url_for('@job?sf_format=atom', true) ?>" />
```

L'attribut `href` de la balise `<link>` reçoit une URL absolue qui est générée à l'aide du second argument du helper `url_for()`.

Générer les informations globales du flux

Le premier objectif de la construction du flux consiste à générer les informations globales de ce dernier. Pour ce faire, l'en-tête actuel du flux doit être remplacé par le code ci-dessous.

Ajout des informations générales du flux ATOM dans le fichier apps/frontend/modules/job/templates/indexSuccess.atom.php

```
<title>Jobeet</title>
<subtitle>Latest Jobs</subtitle>
<link href="<?php echo url_for('@job?sf_format=atom', true) ?>"
rel="self"/>
<link href="<?php echo url_for('@homepage', true) ?>"/>
<updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
strtotime(Doctrine::getTable('JobeetJob')->getLatestPost()->
getCreatedAt())) ?></updated>
<author>
  <name>Jobeet</name>
</author>
<id><?php echo sha1(url_for('@job?sf_format=atom', true)) ?>
</id>
```

Ce template fait usage de la fonction `strtotime()` afin d'obtenir la valeur du champ `created_at` sous la forme d'un timestamp Unix. Pour obtenir la date de création de la dernière offre postée, il suffit de créer la méthode `getLatestPost()` suivante.

Implémentation de la méthode getLatestPost() dans la classe JobeetJobTable du fichier 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();
  }

  // ...
}
```

Il ne reste à présent qu'à générer toutes les entrées du flux correspondantes aux dernières offres publiées sur le site Internet.

Générer les entrées du flux ATOM

Chaque entrée est composée d'un titre, d'un contenu au format HTML, d'un lien, d'un identifiant unique, d'une date de publication et d'un auteur. Toutes ces informations sont bien évidemment issues de la base de données grâce à l'action `index` du module `job`.

Implémentation des entrées du flux ATOM dans le fichier apps/frontend/modules/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', strtotime($job->getCreatedAt())) ?>
      </updated>
      <summary type="xhtml">
       <div xmlns="http://www.w3.org/1999/xhtml">
         <?php if ($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 `sfWebRequest` (`$sf_request`) retourne le serveur courant, qui permet ensuite de construire aisément un lien absolu vers le logo de la société en recherche de nouveaux collaborateurs. La fonction `gmstrftime()` se charge quant à elle de formater une date GMT d'après le paramétrage de la locale du serveur.

Figure 14–2
Affichage du flux ATOM des dernières offres dans le navigateur Safari

Lors du développement de flux de syndication, le débogage de ce dernier peut être complexe avec le navigateur comme seul outil, dans la mesure où ce dernier n'affiche pas la source XML par défaut qui est générée. L'idéal est donc de s'appuyer sur des outils plus pratiques en ligne de commande tels que `curl` ou `wget` qui permettent de récupérer une ressource identifiée par son URL. Le contenu textuel du flux ainsi récupéré devient alors un outil supplémentaire pour apprécier les erreurs et les corriger.

Flux ATOM des dernières offres d'une catégorie

L'un des objectifs de l'application Jobeet est d'aider les internautes à trouver des offres d'emploi ciblées à leur profil. Partant de ce besoin, il s'avère judicieux de produire un flux d'offres d'emploi dédié à chaque catégorie. Disposer d'un flux dynamique pour chaque catégorie a l'avantage de diffuser encore plus de contenus sur Internet mais également de satisfaire les besoins de chaque utilisateur en termes de pertinence de l'information.

Les prochaines sections abordent pas à pas la génération du flux dynamique de chaque catégorie. Le processus de fabrication de ces flux s'échelonne sur 5 étapes successives :

1. mise à jour de la route de la catégorie ;
2. ajout des liens du flux de la catégorie dans les templates ;
3. refactorisation du code de génération des entrées du flux des dernières offres ;
4. simplification du template `indexSuccess.atom.php` du module `job` ;
5. génération du template du flux d'une catégorie.

Pour commencer, il s'agit de mettre à jour la route dédiée de la catégorie afin de rendre le flux publiquement accessible à travers un navigateur web.

Mise à jour de la route dédiée de la catégorie

Pour commencer, la route d'accès au détail d'une catégorie doit être mise à jour afin qu'elle prenne en considération la variable `sf_format` comme le montre le code ci-dessous.

Intégration du support du format ATOM à la route category du fichier de configuration 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)
```

L'URL de la route se termine désormais par une extension bien précise. Il s'agit soit de l'extension `.html` (par défaut) soit de l'extension `.atom`. En fonction de la valeur de celle-ci, Symfony choisira automatiquement la vue correspondante qu'il doit rendre au client.

Mise à jour des liens des flux de la catégorie

Désormais, les deux liens qui pointent vers le flux ATOM doivent être mis à jour. Ces liens figurent respectivement dans les fichiers `indexSuccess.php` et `showSuccess.php` des modules `job` et `category`.

Extrait de code à remplacer dans le fichier 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>
```

Extrait de code à remplacer dans le fichier 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>
```

Factoriser le code de génération des entrées du flux

La dernière étape du parcours consiste à créer le tempate `showSuccess.atom.php`. Le flux de syndication a bien évidemment besoin de la liste des offres d'emploi ; c'est pourquoi il semble opportun de refactoriser le code qui génère les entrées du flux. Il convient donc d'ajouter un nouveau template partiel `_list.atom.php` au projet, qui se charge de générer toutes les entrées du flux. De la même manière qu'avec le format HTML, les templates partiels sont spécifiques au format utilisé.

Contenu du fichier apps/frontend/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',
                 strtotime($job->getCreatedAt())) ?>
      </updated>
    <summary type="xhtml">
     <div xmlns="http://www.w3.org/1999/xhtml">
       <?php if ($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; ?>
```

Simplifier le template indexSuccess.atom.php

Maintenant que le code est proprement isolé dans un template partiel, le template `indexSuccess.atom.php` du module `job` peut à son tour être simplifié en profitant de cette modification.

Contenu du fichier 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?sf_format=atom', true)
?>" rel="self"/>
  <link href="<?php echo url_for('@homepage', true) ?>"/>
  <updated><?php echo gmstrftime('%Y-%m-%dT%H:%M:%SZ',
   strtotime(Doctrine::getTable('JobeetJob')->
             getLatestPost()->getCreatedAt())) ?></updated>
  <author>
    <name>Jobeet</name>
  </author>
  <id><?php echo sha1(url_for('@job?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>
```

Générer le template du flux des offres d'une catégorie

Le template `showSuccess.atom.php` du module `category` est sensiblement le même que celui des dernières offres, à ceci près que les informations globales du flux concernent cette fois-ci la catégorie. Il s'agit donc d'adapter les informations d'en-tête du flux avec celles correspondant à la catégorie demandée. Le code ci-dessous présente le contenu du fichier `showSuccess.atom.php`.

Le fichier 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',
strtotime($category->getLatestPost()->getCreatedAt())) ?>
  </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>
```

Ce template fait appel à la méthode `getLatestPost()` qui renvoie la toute dernière offre postée dans cette catégorie. Cette nouvelle méthode n'existe pas encore et doit donc être implémentée dans la classe `JobeetCategory` comme le montre le morceau de code ci-dessous.

Implémentation de la méthode getLatestPost() de la classe JobeetCategory dans le fichier lib/model/doctrine/JobeetCategory.class.php

```
class JobeetCategory extends BaseJobeetCategory
{
  public function getLatestPost()
  {
    $jobs = $this->getActiveJobs(1);

    return $jobs[0];
  }

  // ...
}
```

La méthode `getActiveJobs()` retourne une collection d'objets `JobeetJob` bien qu'il n'y ait qu'un seul enregistrement récupéré. C'est pour cette raison qu'il faut utiliser la syntaxe `ArrayAccess` sur l'objet `Doctrine_Collection` afin de renvoyer l'objet unitaire. La classe `Doctrine_Collection` implémente également une méthode `getFirst()` qui permet d'obtenir le premier objet de la collection, ce qui revient exactement au même que la syntaxe `ArrayAccess` employée ici.

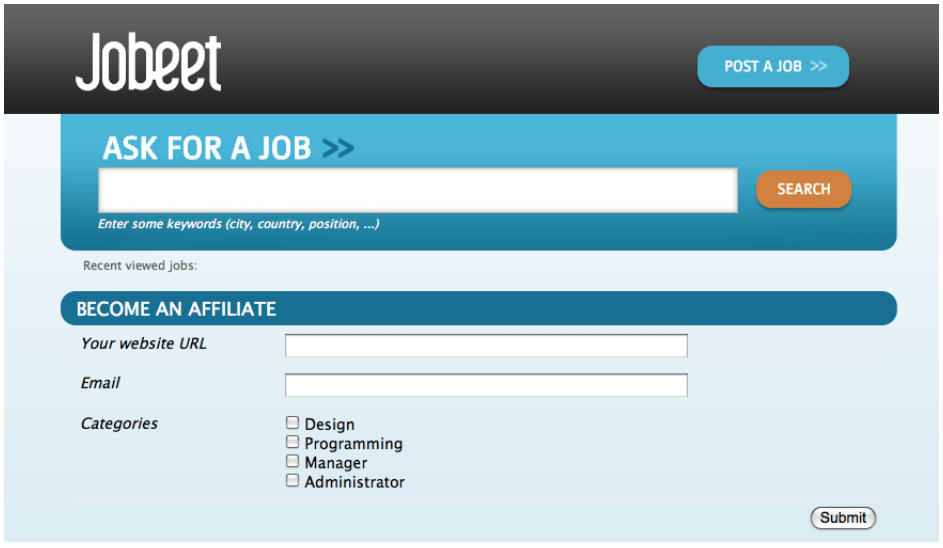
Figure 14–3 Affichage du flux ATOM des offres d'une catégorie dans le navigateur Safari

En résumé…

Comme pour la plupart des fonctionnalités de Symfony déjà décrites, l'ajout de flux de syndication aux applications web s'est effectué sans effort, et ceci grâce au support natif des formats de sortie. Ce chapitre a donc permis de faciliter la vie de l'utilisateur en recherche d'emploi, en lui fournissant un moyen simple et efficace de se tenir directement informé, depuis son navigateur ou son agrégateur favori, des dernières offres d'emploi publiées.

Le chapitre suivant va plus loin dans la manière d'exposer les offres aux internautes, en leur fournissant un service web (*Web Service*)…

chapitre 15


Jobeet
POST A JOB >>
ASK FOR A JOB >>
SEARCH
Enter some keywords (city, country, position, ...)
Recent viewed jobs:
BECOME AN AFFILIATE
Your website URL
Email
Categories
Design
Programming
Manager
Administrator
Submit

Construire des services web

Aujourd'hui, de plus en plus de sites modernes proposent des services web aux développeurs afin que ces derniers soient capables d'intégrer leur contenu aisément dans leurs propres applications web.

La puissance du routage de Symfony ainsi que le support natif des formats de sortie apportent une solution efficace et rapide pour la conception de services web.

MOTS-CLÉS :

- Formats XML, JSON et YAML
- Envoi d'e-mails
- Tests fonctionnels
- REST

Avec l'arrivée des flux de syndication ATOM dans Jobeet, les utilisateurs en recherche d'emploi ont désormais la possibilité d'être tenus informés de la publication de nouvelles offres en temps réel. C'est un excellent début pour améliorer leur confort d'utilisation mais c'est surtout un moyen efficace de diffuser de l'information à travers Internet pour lui garantir une meilleure visibilité.

De l'autre côté, lorsqu'une nouvelle offre d'emploi est postée, il convient idéalement de lui faire profiter de la meilleure exposition possible. En effet, plus l'annonce est syndiquée sur un réseau de petits sites, et plus elle aura de chance d'attirer les meilleurs profils. C'est tout le pouvoir de la « longue traîne » (*long tail*), ce qui signifie ici que même les petits sites Internet ou weblogs représentent potentiellement une part non négligeable dans la réussite de l'offre. Grâce aux services web qui seront développés tout au long de ce chapitre, les partenaires et les affiliés seront capables d'afficher sur leurs sites web les toutes dernières offres d'emploi publiées.

Concevoir le service web des offres d'emploi

L'objectif de ce quinzième chapitre est de réaliser pas à pas un service web à destination des développeurs. Ce service sera accessible au moyen d'une interface de programmation applicative (API) simple reposant sur l'appel d'URLs et sur la récupération d'informations dans différents formats de sortie tels que le XML, le JSON ou bien encore le YAML. La conception de ce service web se déroule en plusieurs étapes successives qui nécessitent entre autres de déclarer la route de l'API, d'implémenter l'action à exécuter ou bien encore de construire les templates relatifs à chaque format de sortie demandé. Pour commencer en douceur, il convient de préparer quelques jeux de données initiales.

Préparer des jeux de données initiales des affiliés

Le troisième chapitre a été l'occasion d'établir et de découvrir le schéma de description de la base de données. Ce dernier définit deux entités de modèle qui n'ont pas encore été exploitées jusqu'à présent : `JobeetAffiliate` et `JobeetCategoryAffiliate`. La table `jobeet_affiliate` stocke les informations du site Internet partenaire tandis que la relation `jobeet_category_affiliate` se contente de garder en mémoire la liste des catégories auxquelles est affilié ce dernier.

Comme avec les catégories et les offres d'emploi, il est bon de démarrer le développement d'une nouvelle fonctionnalité en préparant quelques

jeux de données initiales. Le code ci-dessous déclare deux sites Internet partenaires et leur associe à chacun une liste de catégories.

Jeu de données initiales du fichier 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]
```

Comme le montre le contenu de ce fichier YAML, la création d'enregistrements pour une relation « many-to-many » est aussi simple que définir un tableau dont les valeurs sont les noms des enregistrements des entités en relation. Le contenu du tableau de catégories correspond au nom des objets définis dans les fichiers de données. Les objets peuvent ainsi être liés entre eux à divers endroits et dans différents fichiers à condition qu'ils aient tous bien été déclarés en premier.

Pour des raisons de simplification des tests, les jetons de chaque affilié sont codés en dur dans le fichier YAML. Lorsque le site sera en production, ces derniers devront bien évidemment être générés automatiquement au moment où l'utilisateur postulera pour un compte.

Implémentation de la méthode preValidate() dans la classe JobeetAffiliate du fichier lib/model/doctrine/JobeetAffiliate.class.php

```
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function preValidate($event)
  {
    $object = $event->getInvoker();

    if (!$object->getToken())
    {
      $object->setToken(sha1($object->getEmail().rand(11111, 99999)));
    }
  }

  // ...
}
```

La méthode `preValidate()` d'un objet Doctrine est toujours exécutée avant que l'objet ne soit sérialisé en base de données. Elle a pour rôle de contrôler que l'objet courant est valide et qu'il peut être sauvegardé en base de données. La méthode `getInvoker()` de l'objet `Doctrine_Event`, lui-même passé en paramètre de la méthode `preValidate()`, retourne l'objet `JobeetAffiliate`. La condition qui lui succède vérifie si le jeton est déjà défini. S'il ne l'est pas encore, il est alors généré automatiquement à partir de l'adresse e-mail et d'une valeur aléatoire. Grâce à ce mécanisme, le champ `token` de la table `jobeet_affiliate` ne peut rester nul, et donc l'objet reste valide.

Au final, il ne reste plus qu'à charger les jeux de données initiales dans la base de données au moyen de la commande Symfony `doctrine:data-load`.

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

Construire le service web des offres d'emploi

Déclaration de la route dédiée du service web

Comme toujours, la première bonne pratique à mettre en œuvre lorsqu'une nouvelle fonctionnalité est sur le point d'être implémentée, est de déclarer une route dédiée pour la rendre accessible. Dans le cas présent, il s'agit de définir une URL propre à chaque affilié qui fait usage de la variable spéciale `sf_format` vue au cours du précédent chapitre. Le jeton qui vient tout juste d'être créé pour le modèle `JobeetAffiliate` sert effectivement à rendre la route dépendante de l'affilié.

La route ci-dessous implémente donc ce jeton ainsi que la variable spéciale `sf_format` afin de déterminer dans quel format les informations doivent être délivrées par l'API.

Déclaration de la route api_jobs dans le fichier 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)
```

Cette route se termine par la variable `sf_format` qui, d'après les restrictions qui lui sont appliquées, peut prendre l'une des valeurs parmi `xml`, `json` ou bien `yaml`.

Implémenter la méthode getForToken() de l'objet JobeetJobTable

De plus, cette route a besoin d'une méthode `getForToken()` qui est appelée lorsque l'action récupère la collection d'objets en relation. Dans la mesure où il faut s'assurer que l'affilié est bien activé, le comportement par défaut de la route doit être surchargé.

Implémentation de la méthode getForToken() dans le fichier lib/model/doctrine/JobeetJobTable.class.php

```
class JobeetJobTable extends Doctrine_Table
{
  public function getForToken(array $parameters)
  {
    $affiliate = Doctrine::getTable('JobeetAffiliate')
         ->findOneByToken($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();
  }

  // ...
}
```

La méthode `getForToken()` se charge de récupérer un objet `JobeetAffiliate` à partir de son jeton unique. Si le jeton n'existe pas dans la base de données, alors une exception de type `sfError404Exception` est levée afin d'être automatiquement convertie en réponse 404 par Symfony. Lancer ce type d'exception est la manière la plus simple de générer des pages d'erreur 404 depuis une classe de modèle.

Il faut aussi remarquer que `getForToken()` fait appel à la méthode virtuelle `findOneByToken()` de la classe `JobeetAffiliateTable`. Doctrine permet de récupérer un objet unique d'une table en utilisant la méthode `findOneBy*()` où * est le nom du champ dans la table qui sert de critère de restriction (clause `WHERE` de la requête SQL). Ces méthodes virtuelles sont automatiquement générées par Doctrine à l'aide de l'implémentation de la méthode magique `__call()` de PHP. C'est grâce à cette dernière qu'il est rendu possible d'appeler des méthodes non définies explicitement dans la classe de l'objet.

Implémenter la méthode getActiveJobs() de l'objet JobeetAffiliate

D'autre part, la méthode `getForToken()` utilise une nouvelle méthode `getActiveJobs()` afin de retourner la liste des offres actives courantes.

Implémentation de la méthode getActiveJobs() dans la classe JobeetAffiliate du fichier 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());

    $q = Doctrine::getTable('JobeetJob')
      ->addActiveJobsQuery($q);

    return $q->execute();
  }

  // ...
}
```

La dernière étape consiste à mettre en place l'action de l'API et ses templates. Pour ce faire, il suffit de générer un nouveau module à l'aide de la commande `generate:module`.

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

Dans la mesure où l'action `index` par défaut n'est pas utile au reste de l'application, elle peut être supprimée en toute sécurité de la classe d'actions, ainsi que son template associé `indexSuccess.php`.

Développer le contrôleur du service web

Implémenter l'action executeList() du module api

Tous les formats de sortie de la route `api_jobs` partagent la même action `list` du module `api`. En sachant cela, il convient seulement d'implémenter la méthode `executeList()` puis de construire les templates propres à chaque format de sortie définis pour la variable `sf_format`.

Le corps de cette méthode ne pose aucune difficulté particulière dans la mesure où cette dernière se charge de récupérer la liste des objets `JobeetJob` à partir de la route appelée, puis de représenter chaque objet sous la forme d'un tableau avant de stocker ce dernier dans un autre tableau passé au template correspondant.

Implémentation de la méthode executeListe() dans le fichier 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());
  }
}
```

Implémenter la méthode asArray() de JobeetJob

Au lieu de passer un tableau d'objets `JobeetJob` aux templates comme c'est le cas d'habitude, l'action transmet un tableau de chaînes de caractères. Comme l'action est partagée par trois templates différents, la logique métier de traitement des valeurs a été mutualisée ailleurs dans la méthode `JobeetJob::asArray()`.

Implémentation de la méthode asArray() dans la classe JobeetJob du fichier 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 code métier de l'action est à présent complètement implémenté, et la prochaine étape consiste alors à développer le template `listSuccess.php` pour chaque format de sortie désiré.

Construction des templates XML, JSON et YAML

Cette section aborde la génération des templates pour les trois formats de sortie possibles. Les templates pour le format XML et JSON sont relativement simples à comprendre et à mettre en œuvre, c'est pourquoi il n'y aura que très peu d'explications à leur égard. En revanche, le format YAML nécessitera d'approfondir quelques notions subtiles comme la gestion des erreurs en fonction des environnements.

Le format XML

Le format XML est aussi simple à gérer que le HTML puisqu'il s'agit de créer un nouveau template contenant la génération du code XML à renvoyer au client. Dans le cadre de Jobeet, il s'agit d'aboutir à un fichier XML similaire à la maquette suivante :

```
<?xml version="1.0" encoding="utf-8"?>
<jobs>
  <job url="http://www.jobeet.org/en/job/extreme-sensio/paris-
france/2/web-designer">
    <category>design</category>
    <type>part-time</type>
    <logo>http://www.jobeet.org/uploads/jobs/extreme-
sensio.gif</logo>
    <location>Paris, France</location>
    <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.
    </description>
    <how_to_apply>Send your resume to fabien.potencier [at]
sensio.com</how_to_apply>
    <expires_at>2009-05-16</expires_at>
  </job>
  <!-- ... ->
</jobs>
```

Ce gabarit XML ne pose aucune difficulté à générer. En effet, les noms des balises correspondent aux clés du tableau PHP renvoyé par la méthode `asArray()`. Seules quelques lignes de code PHP suffisent à construire un tel template comme le montre le code ci-dessous.

Contenu du 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>
```

Seulement deux instructions `foreach()` suffisent au parcours du tableau `$jobs`. La première itère sur la liste des offres tandis que la seconde se charge de traverser les propriétés de chacune d'entre elles afin d'en générer les bons couples balise/contenu.

Le format JSON

JSON (*JavaScript Object Notation*), est un format de données standard dérivé de la notation des objets du langage ECMAScript, et qui a pour objectif premier de structurer de l'information à l'aide d'une syntaxe simple et lisible par les développeurs. Le format JSON permet de décrire différents types de structures de données comme les objets, les tableaux, les entiers, les chaînes de caractères, les booléens…

L'API de Jobeet supporte nativement le format JSON. Il s'agit donc à présent de développer le template correspondant capable de générer une réponse au format JSON identique au code ci-dessous.

```
[
{
  "url": "http://www.jobeet.org/en/job/extreme-sensio/
          ➥ paris-france/2/web-designer",
  "category": "design",
  "type": "part-time",
  "logo": "http:\/\/www.jobeet.org\/uploads\/jobs\/
          ➥ extreme-sensio.gif",
  "location": "Paris, France",
  "description": "\tLorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do \n\t\teiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut \n\t\tenim ad minim veniam,
quis nostrud exercitation ullamco laboris \n\t\tnisi ut aliquip
ex ea commodo consequat. Duis aute irure dolor \n\t\tin
reprehenderit in.\n\t\tVoluptate velit esse cillum dolore eu
fugiat nulla pariatur. \n\t\tExcepteur sint occaecat cupidatat
non proident, sunt in culpa \n\t\tqui officia deserunt mollit
anim id est laborum.",
```

```
  "how_to_apply": "Send your resume to fabien.potencier [at]
sensio.com",
  "0": "2009-05-16"
}
]
```

La génération de ce type de résultat JSON est réalisée au moyen des quelques lignes de code PHP qui suivent, et notamment grâce à la fonction native `json_encode()` du langage PHP.

Contenu du template 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 $nbl = count($job); $j = 0; foreach ($job as $key =>
$value): ++$j ?>
  "<?php echo $key ?>": <?php echo json_encode($value).($nbl ==
$j ? '' : ',') ?>

<?php endforeach; ?>
}<?php echo $nb == $i ? '' : ',' ?>

<?php endforeach; ?>
]
```

Le format YAML

Paramétrer les caractéristiques de la réponse

Symfony configure automatiquement certains paramètres tels que les en-têtes de type de contenu (*Content-Type*) ou bien encore la désactivation du layout pour les formats de sortie standards intégrés au framework. Le format YAML ne fait pas partie de la liste des formats standards supportés nativement, c'est pourquoi les en-têtes HTTP ainsi que la suppression du layout doivent être gérés manuellement dans les actions.

```
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;
    }
  }
}
```

L'instruction `switch()` ci-dessus teste la valeur du format de sortie demandé. Si celui-ci répond à la valeur `yaml` alors le layout est désactivé pour ne pas décorer le template de l'action, et l'en-tête HTTP `Content-Type` de la réponse est fixé à la valeur `text/yaml`.

Construire le template de génération de la sortie YAML

Le template au format YAML n'est guère plus complexe à mettre en œuvre que les deux précédents dans la mesure où Symfony fournit tous les outils nécessaires à la conversion de tableaux PHP en chaînes de caractères YAML.

Contenu du template 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 l'on tente d'appeler le service web avec un jeton invalide, une page d'erreur 404 au format XML ou JSON sera levée. Or, le format YAML n'est pas un format supporté nativement par le framework. Il en résulte alors que ce dernier ne sait pas quel template rendre au client. La section suivante explique comment définir les pages d'erreur 404 pour le format YAML en tenant compte des environnements de développement et de production.

Génération des pages d'erreur 404 en fonction de l'environnement

À chaque fois qu'un nouveau format est créé, un template d'erreur associé doit également être préparé. Symfony se servira en effet de ce template pour rendre les pages d'erreur 404 ainsi que toutes les autres exceptions levées. Or, le rendu d'une erreur ou d'une exception n'est pas le même selon que l'application est exécutée en environnement de développement ou de production.

De ce fait, il est nécessaire de gérer ces deux cas de figure en fournissant deux templates distincts : `config/error/exception.yaml.php` pour le débogage et `config/error/error.yaml.php` pour la production. Surcharger les pages par défaut d'erreur 404 ou d'exception de Symfony revient simplement à créer un fichier dans le répertoire `config/error/`.

Contenu du template d'affichage des exceptions du format YAML dans le fichier 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) ?>
```

Contenu du template d'affichage des erreurs 404 du format YAML dans le fichier config/error/error.yaml.php

```
<?php echo sfYaml::dump(array(
  'error'       => array(
    'code'      => $code,
    'message'   => $message,
))) ?>
```

La création de ces deux templates d'erreur ne suffit pas pour pouvoir les tester. Il manque en effet la mise en place d'un layout dédié au format YAML.

Contenu du layout propre au format YAML dans le fichier apps/frontend/templates/layout.yaml.php

```
<?php echo $sf_content ?>
```

Figure 15–1
Récupération d'une page d'erreur 404 au format YAML en environnement de développement

Le service web est maintenant entièrement fonctionnel et prêt à l'emploi. Cependant, il ne sera mis en production qu'après lui avoir fait subir quelques séries de tests fonctionnels pour en valider le bon comportement.

Écrire des tests fonctionnels pour valider le service web

Comme toujours, les tests fonctionnels nécessitent des jeux de données. La première étape consiste donc à copier les fichiers de données initiales du modèle `JobeetAffiliate` depuis le répertoire `data/fixtures` vers `test/fixtures`. Une fois cette manipulation accomplie, le contenu du fichier autogénéré `apiActionsTest.php` peut être remplacé par le code suivant :

Scénarios de tests fonctionnels du module api dans le fichier 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')->checkElement('job', 32)->

  info('3 - The web service supports the JSON format')->
  get('/api/sensio_labs/jobs.json')->
  with('request')->isFormat('json')->
  with('response')->contains('"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')->
```

```
    contains('category: Programming')->
  end()
;
```

L'ensemble de cette suite de tests fonctionnels est suffisamment explicite pour ne pas être détaillée davantage dans la mesure où la plupart des concepts ici présents ont déjà été expliqués au chapitre 9. Il faut cependant remarquer l'utilisation de deux nouvelles méthodes, `isFormat()` et `contains()`, qui contrôlent respectivement le format de sortie attendu et le contenu de la réponse lorsque celle-ci ne peut être analysée à l'aide du DOM (et de sélecteurs CSS 3). Enfin, la méthode `isHeader()` du test numéro 4 s'assure que la valeur de l'en-tête de la réponse correspond bien à la chaîne `text/yml; charset=utf-8`.

Formulaire de création d'un compte d'affiliation

Le service web est enfin prêt à être consommé par les sites Internet partenaires. Cependant, l'utilisation du service oblige l'affilié à s'enregistrer auprès de l'application Jobeet afin de se voir délivrer un jeton unique. Cette opération est bien évidemment gratuite et réalisable via un court formulaire d'inscription. La section qui suit décrit en cinq étapes successives comment mettre en œuvre et tester cette nouvelle fonctionnalité.

Déclarer la route dédiée du formulaire d'inscription

Comme toujours, l'activation d'une nouvelle route dédiée pour la ressource est la première étape à mettre en œuvre. Il s'agit ici de déclarer une nouvelle collection de routes Doctrine capable de gérer un ensemble d'actions nécessaires au bon fonctionnement du mécanisme d'inscription des affiliés.

Déclaration de la route affiliate dans le fichier apps/frontend/config/routing.yml

```
affiliate:
  class:   sfDoctrineRouteCollection
  options:
    model: JobeetAffiliate
    actions: [new, create]
    object_actions: { wait: get }
```

Le code ci-dessus définit une collection de routes Doctrine dans laquelle figure une nouvelle option de configuration : `actions`. Dans la mesure où le processus d'inscription n'a pas besoin des sept actions par défaut définies dans la route, l'option `actions` force la route à n'être active que pour les

actions `create` et `new`. La route additionnelle `wait` sera utilisée pour donner des feedbacks sur son compte à l'affilié en attente de validation.

Générer un module d'amorçage

La seconde étape traditionnelle dans ce processus consiste à générer un module dédié pour cette nouvelle fonctionnalité. Bien évidemment, il convient de faire usage de la commande `doctrine:generate-module` pour accomplir cette tâche.

```
$ php symfony doctrine:generate-module frontend affiliate
JobeetAffiliate --non-verbose-templates
```

Construction des templates

La tâche `doctrine:generate-module` génère les sept actions classiques par défaut ainsi que tous leurs templates correspondants. Tous les templates du répertoire `templates/` peuvent être supprimés à l'exception des fichiers `_form.php` et `newSuccess.php`. Pour ces fichiers restants, leur contenu respectif doit être remplacé par ceux qui suivent.

Contenu du fichier apps/frontend/modules/affiliate/templates/newSuccess.php

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

<h1>Become an Affiliate</h1>

<?php include_partial('form', array('form' => $form)) ?>
```

Contenu du fichier 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>
```

Le template `waitSuccess.php` peut à son tour être créé comme le présente le code ci-dessous.

Contenu du fichier 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>
Last, change the link in the footer to point to the affiliate
module:

// apps/frontend/templates/layout.php
<li class="last">
  <a href="<?php echo url_for('@affiliate_new') ?>">Become an
affiliate</a>
</li>
```

Les templates sont désormais prêts. Il ne leur manque plus que l'implémentation de leur action associée pour rendre le tout fonctionnel.

Implémenter les actions du module affiliate

La plupart du code autogénéré dans le fichier `actions.class.php` n'est pas utile au reste de l'application. De ce fait, tout le code de ce fichier peut être retiré à l'exception des méthodes `executeNew()`, `executeCreate()`, et `processForm()`. L'URL de redirection de l'action `processForm()` doit quant à elle être modifiée.

Modification de l'URL de redirection dans le fichier apps/frontend/modules/affiliate/actions/actions.class.php

```
$this->redirect($this->generateUrl('affiliate_wait',
$jobeet_affiliate));
```

De son côté, l'action `wait` reste triviale puisqu'elle n'implémente aucune logique particulière ni ne passe quoi que ce soit à sa vue correspondante.

Implémentation de la méthode executeWait() dans le fichier apps/frontend/modules/affiliate/actions/actions.class.php

```
public function executeWait(sfWebRequest $request)
{
}
```

Le partenaire ne peut choisir son propre jeton ni ne peut activer son compte immédiatement. Pour remplir ce besoin fonctionnel, il est nécessaire de personnaliser le formulaire `JobeetAffiliateForm`.

Configuration du formulaire d'inscription dans le fichier lib/form/doctrine/JobeetAffiliateForm.class.php

```
class JobeetAffiliateForm extends BaseJobeetAffiliateForm
{
  public function configure()
  {
    unset($this['is_active'], $this['token'],
$this['created_at'], $this['updated_at']);

    $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));
  }
}
```

Le framework de formulaires supporte les relations *many-to-many* avec n'importe quelle colonne d'une table. Par défaut, ce type de relation est rendu possible à l'aide d'une liste déroulante multiple grâce au widget `sfWidgetFormChoice`. Le chapitre 10 a d'ailleurs montré comment le rendu HTML final d'un widget `sfWidgetFormChoice` peut être modifié au moyen de l'option de configuration `expanded`.

De plus, comme les adresses e-mails et les URLs ont tendance à être plus longues que la taille par défaut du champ `input` généré, les attributs HTML à appliquer par défaut peuvent être paramétrés grâce à la méthode `setAttribute()` du widget.

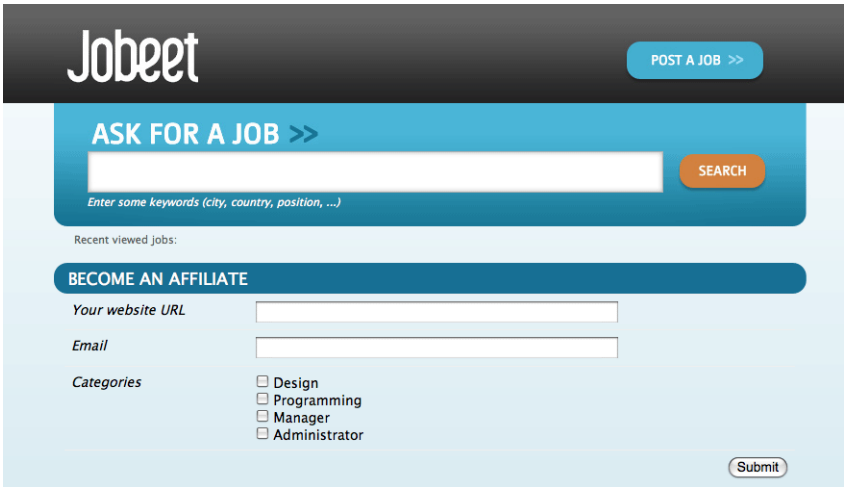


Figure 15–2 Rendu final du formulaire de création de compte à l'API de Jobeet

Tester fonctionnellement le formulaire

La dernière étape consiste comme toujours à écrire quelques tests fonctionnels pour s'assurer que la page se comporte correctement. Les tests autogénérés du module `affiliate` sont à remplacer par la suite de tests fonctionnels ci-après.

Scénarios de tests fonctionnels du module affiliate dans le fichier 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::getTable('JobeetCategory')
          ->findOneBySlug('programming')->getId()),
  )))->
  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')
;
```

Développer l'interface d'administration des affiliés

Le service web est maintenant actif et les développeurs peuvent s'inscrire à l'API. Or, l'inscription à l'API de Jobeet nécessite une activation manuelle par l'administrateur du site. Il convient donc de développer une interface d'administration propice à la gestion des affiliés dans l'application backend. Cette dernière permettra à l'administrateur d'activer, de désactiver ou bien encore de supprimer un compte développeur.

Générer le module d'administration affiliate

La première étape de conception de cette interface d'administration consiste à générer le squelette fonctionnel du module de gestion des affiliés. Bien évidemment, il s'agit de ne pas réinventer la roue et de s'appuyer à nouveau sur le générateur de backoffice de Symfony.

```
$ php symfony doctrine:generate-admin backend JobeetAffiliate --
-module=affiliate
```

Afin de faciliter l'accès à ce module aux administrateurs, le menu principal de navigation doit accueillir un nouveau lien. Ce dernier est défini dans le layout de l'application comme le montre l'exemple de code suivant.

Contenu du fichier apps/backend/templates/layout.php

```
<li>
  <a href="<?php echo url_for('@jobeet_affiliate') ?>">
    Affiliates - <strong><?php echo
Doctrine::getTable('JobeetAffiliate')->countToBeActivated()
?></strong>
  </a>
</li>
```

Ce nouveau lien fait appel à une nouvelle méthode `countToBeActivated()` de l'objet `JobeetAffiliateTable` qui se charge de retourner le nombre de comptes affiliés en attente de validation. L'implémentation de cette méthode est décrite dans le code ci-après.

Implémentation de la méthode countToBeActivated() dans la classe JobeetAffiliateTable du fichier 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();
  }

  // ...

}
```

Paramétrer le module affiliate

Les seules véritables actions nécessaires dans ce module d'administration correspondent à l'activation ou la désactivation des comptes. De ce fait, la vue *list* peut être simplifiée grâce à la configuration suivante. Elle surcharge les paramètres par défaut de la configuration actuelle, et lui ajoute deux liens supplémentaires pour activer ou désactiver un compte partenaire.

Configuration du module de gestion des partenaires dans le fichier 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]
```

Implémenter les nouvelles fonctionnalités d'administration

Afin de rendre les administrateurs plus productifs et plus efficaces dans leurs tâches de gestion de l'application, il semble judicieux de changer les filtres par défaut pour n'afficher que les comptes affiliés en attente de validation. Cette opération est très simple à mettre en œuvre puisqu'il s'agit de redéfinir la méthode par défaut `getFilterDefaults()` de la classe autogénérée `affiliateGeneratorConfiguration`.

Redéfinition de la méthode getFilterDefaults du fichier apps/backend/modules/affiliate/lib/affiliateGeneratorConfiguration.class.php

```
class affiliateGeneratorConfiguration extends
BaseAffiliateGeneratorConfiguration
{
  public function getFilterDefaults()
  {
    return array('is_active' => '0');
  }
}
```

Il ne reste finalement plus qu'à implémenter le code relatif aux actions d'activation et de désactivation de comptes développeur. La vue *list* déclare ces actions aussi bien de manière unitaire (sur chaque objet) que sur un ensemble d'enregistrements sélectionnés dans le tableau. Le code ci-dessous donne l'intégralité des nouvelles méthodes implémentées dans la classe d'actions du module `affiliate`.

Implémentation des actions d'activation et de désactivation de comptes partenaires dans le fichier 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');
  }
}
```

Certains passages de ce code sont présentés en exergue et montrent l'utilisation des méthodes `activate()` et `deactivate()` de l'objet `JobeetAffiliate` qui respectivement activent et désactivent le compte affilié. Ces deux nouvelles méthodes n'existent pas encore dans le projet et doivent être implémentées afin de rendre l'interface de gestion entièrement fonctionnelle.

Implémentation des methods activate() et deactivate() de la classe JobeetAffiliate dans le fichier 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();
  }

  // ...
}
```

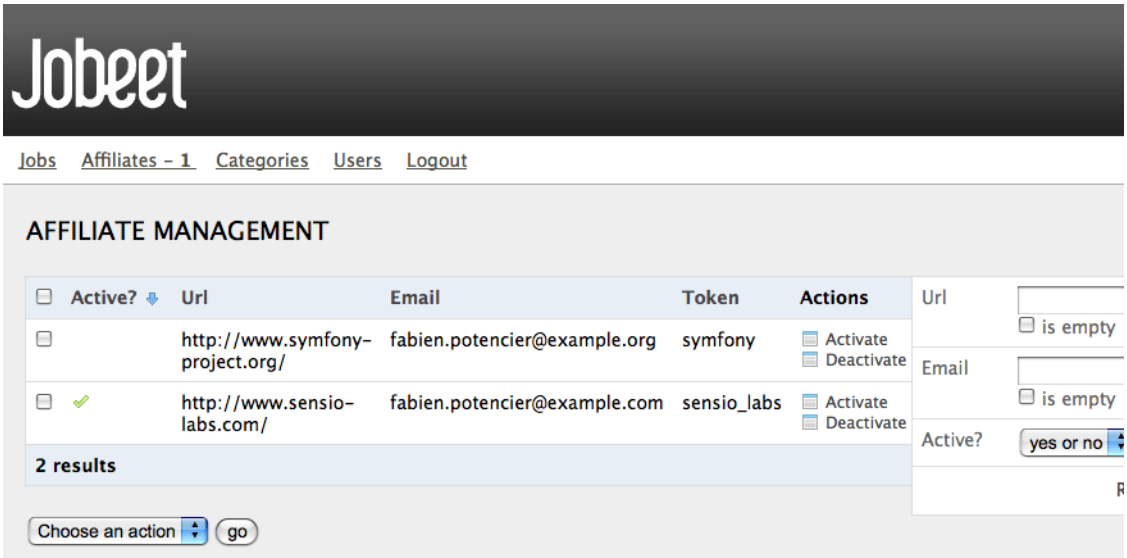


Figure 15–3 Rendu final de l'interface d'administration des comptes affiliés

C'est tout pour ce nouveau module. En seulement quelques minutes, l'application Jobeet dispose d'un module d'administration pour gérer les comptes affiliés qui seront ainsi capables de consommer le service web mis à leur disposition. Bien que ce module soit complètement fonctionnel, il n'en demeure pas moins qu'il lui manque une fonctionnalité primordiale lors de l'activation d'un compte affilié : la notification par e-mail. En effet, le système n'envoie pour l'instant aucune notification à l'utilisateur pour l'informer que son compte a été validé.

La section suivante clôture ce chapitre en expliquant pas à pas comment intégrer un envoi d'e-mail très simple à l'aide du composant `Zend_Mail` du Zend Framework.

Envoyer des e-mails avec Zend_Mail

Lorsqu'un administrateur active le compte d'un affilié, un e-mail devrait automatiquement lui être adressé. Celui-ci lui confirmerait son inscription en lui attribuant son jeton unique afin qu'il puisse consommer le service web. Le langage PHP dispose d'un grand nombre d'excellentes librairies d'envoi d'e-mails comme SwitfMailer, Zend_Mail ou bien encore ezcMail.

Le chapitre suivant aura recours à des composants du Zend Framework pour faciliter la mise en place d'un outil de recherche. C'est en effet dans le but de conserver une certaine cohérence tout au long du projet que seront utilisés ici les composants Zend_Mail et Zend_Search du Zend Framework.

Installer et configurer le framework Zend

La librairie `Zend_Mail` est un composant à part entière du framework Zend. Pour répondre aux besoins technologiques de ce chapitre et des suivants, il n'est pas utile d'installer l'intégralité du framework Zend. Seuls quelques composants de celui-ci ont leur utilité pour le projet. Heureusement, la plupart des paquets du Zend Framework sont suffisamment découplés, autonomes et indépendants les uns des autres pour pouvoir être récupérés individuellement.

La première étape consiste donc à recréer la structure allégée du Zend Framework en commençant par créer un répertoire `Zend` dans le répertoire `lib/vendor/` du projet. Puis les dossiers et fichiers suivants du Zend Framework doivent être copiés dans ce nouveau répertoire :

- `Exception.php`
- `Loader/`
- `Loader.php`
- `Mail/`
- `Mail.php`
- `Mime/`
- `Mime.php`
- `Search/`

Le répertoire `Loader/` contient le composant de chargement des classes du Zend Framework à la demande. Les paquets `Mail/` et `Mime/` contiennent les classes nécessaires à l'envoi d'e-mails avec ou sans pièces jointes tandis que le composant `Search/` aura la charge d'indexer et de retrouver du contenu pour le moteur de recherche de Jobeet.

Maintenant que les composants du Zend Framework sont intégrés au projet, il ne reste plus qu'à indiquer à Symfony comment il doit charger les classes. C'est en réalité le composant `Zend_Loader` qui a la responsabilité de charger dynamiquement les classes du framework Zend. Le code ci-dessous présente de quelle manière ce composant doit être initialisé depuis la classe de configuration globale du projet.

Implémentation de la méthode registerZend() dans la classe ProjectConfiguration du fichier 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.php';
    Zend_Loader::registerAutoload();
    self::$zendLoaded = true;
  }

  // ...
}
```

La classe `ProjectConfiguration` est dotée à présent d'un nouvel attribut booléen, protégé et statique qui détermine si les classes du Zend Framework ont été chargées automatiquement ou pas. La méthode `registerZend()`, quant à elle, a pour mission d'ajouter le répertoire `lib/vendor` à la liste des chemins dans lesquels PHP tentera de trouver et d'importer les classes demandées, mais aussi et surtout de charger et d'initialiser le chargement automatique de classes du framework Zend.

Implémenter l'envoi d'un e-mail à l'activation du compte de l'affilié

La dernière étape consiste à implémenter la logique métier de l'envoi de l'e-mail à destination de l'affilié lors de l'activation de son compte par un administrateur. L'envoi du courrier électronique est laissé à la charge du contrôleur, c'est pour cette raison qu'il trouve naturellement sa place dans le corps de la méthode `executeListActivate()`.

Implémentation de l'envoi d'e-mails dans la méthode executeListActivate() du fichier 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
    ProjectConfiguration::registerZend();
    $mail = new Zend_Mail();
    $mail->setBodyText(<<<EOF
Your Jobeet affiliate account has been activated.

Your token is {$affiliate->getToken()}.

The Jobeet Bot.
EOF
);
    $mail->setFrom('jobeet@example.com', 'Jobeet Bot');
    $mail->addTo($affiliate->getEmail());
    $mail->setSubject('Jobeet affiliate token');
    $mail->send();

    $this->redirect('@jobeet_affiliate');
  }

  // ...
}
```

Le principe de fonctionnement du code mis en avant est simple. L'appel à la méthode statique `registerZend()` permet d'importer et d'initialiser la classe de chargement automatique des composants du framework Zend. Une fois cette opération accomplie, un nouvel objet `Zend_Mail` est instancié afin de préparer l'e-mail qui est finalement envoyé à son destinataire à l'appel de la méthode `send()`. Pour que l'envoi de l'e-mail fonctionne correctement, l'adresse e-mail fictive `jobeet@example.com` doit être remplacée par une adresse réelle.

Bien sûr, il ne s'agit ici que d'un exemple minimaliste de génération d'e-mail avec cette librairie. Le composant `Zend_Mail` recèle bien d'autres atouts comme l'envoi d'e-mails au format HTML, le support des pièces jointes, la manipulation des messages via les protocoles SMTP et POP3… La documentation de `Zend_Mail` ainsi que de nombreux exemples de mise en pratique sont présentés sur le site officiel du framework Zend.

▸ http://framework.zend.com/

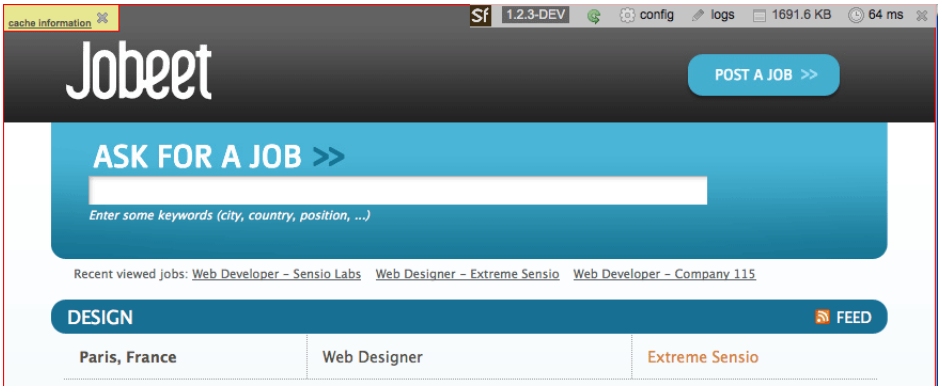
En résumé…

Grâce à l'architecture REST de Symfony, l'implémentation de services web dans les projets se voit grandement simplifiée. Bien que le code écrit dans ce chapitre corresponde uniquement à un service web accessible en lecture, vous disposez à présent de toutes les connaissances suffisantes pour développer votre propre service web interrogeable en lecture et en écriture.

L'implémentation du formulaire de création d'un compte affilié dans les applications frontend et backend a été particulièrement facilitée dans la mesure où vous êtes maintenant familier avec tout le processus d'ajout de nouvelles fonctionnalités à un projet Symfony.

Le chapitre suivant implémente la toute dernière fonctionnalité majeure du site Internet de Jobeet – le moteur de recherche – développée à l'aide du composant `Zend_Search_Lucene` du framework Zend…

chapitre 16


cache information
Sf 1.2.3-DEV config logs 1691.6 KB 64 ms
Jobeet
POST A JOB >>
ASK FOR A JOB >>
Enter some keywords (city, country, position, ...)
Recent viewed jobs: Web Developer - Sensio Labs Web Designer - Extreme Sensio Web Developer - Company 115
DESIGN
FEED
Paris, France
Web Designer
Extreme Sensio

Déployer un moteur de recherche

MOTS-CLÉS :

- Indexation avec Zend_Search_Lucene
- Transactions SQL avec Doctrine
- Tests unitaires
- Tâches automatisées

L'une des fonctionnalités les plus difficiles à mettre en œuvre dans une application web dynamique est le moteur de recherche, dans la mesure où il doit généralement être capable de rechercher dans l'intégralité du contenu. Quelques outils d'indexation de contenu open source existent dont le plus connu, Lucene, dispose d'un composant écrit en PHP pour le framework Zend. C'est celui-ci qui sera implémenté pour Jobeet dans la suite de ce chapitre.

L'application Jobeet dispose de tout l'équipement nécessaire pour assurer la création de nouveaux contenus, ainsi que la diffusion de ces derniers à travers Internet à l'aide des flux ATOM et de son service web. Néanmoins, il manque une fonctionnalité essentielle au site Internet : un moteur de recherche. Ce moteur de recherche doit en effet permettre aux utilisateurs de faciliter leur recherche d'offres d'emploi en saisissant des mots-clés pertinents. L'objectif de ce chapitre est de mettre en place pas à pas un moteur de recherche basé sur la solution d'indexation Lucene.

Découverte de la librairie Zend_Search_Lucene

Rappels historiques au sujet de Symfony

Avant de plonger tête la première dans le développement du moteur de recherche, un bref rappel de l'historique de Symfony semble opportun. L'équipe du projet Symfony s'efforce depuis toujours de prôner les meilleures pratiques de développement, telles que les tests et le refactoring du code, et par la même occasion de les mettre en œuvre au sein du framework lui-même. L'une des principales devises du framework est par exemple de ne pas réinventer la roue (*do not reinvent the wheel*), et c'est pour cette raison que Symfony est né il y a quatre ans en embarquant deux projets Open Source matures : Mojavi et Propel.

Aujourd'hui encore, lorsqu'il y a un nouveau problème à résoudre, l'équipe de Symfony se pose toujours la question de savoir s'il existe déjà ou non une quelconque implémentation récupérable qui répond au besoin, avant de coder quoique ce soit depuis zéro.

Présentation de Zend Lucene

L'objectif de ce chapitre est d'ajouter un moteur de recherche à l'application Jobeet, et heureusement le framework Zend fournit une excellente librairie nommée Zend_Search_Lucene, qui est un portage en PHP du célèbre projet Open Source Java Lucene. Au lieu de créer un nouveau moteur de recherche pour Jobeet, ce qui est une tâche particulièrement complexe et chronophage, il sera plus pertinent de réutiliser le composant Zend Lucene. La documentation de Zend_Search_Lucene présente sur le site officiel du framework Zend décrit cette librairie de la manière suivante :

Zend Lucene est un moteur de recherche de contenus principalement textuels écrit entièrement en PHP 5. Comme il sauvegarde son index sur le système de fichiers local et qu'il ne requiert aucun serveur de base de données, il peut ajouter des capacités de recherche à la plupart des sites Internet conçus en PHP 5. Zend_Search_Lucene supporte les fonctionnalités suivantes :

- recherche triée par pertinence : les meilleurs résultats sortent en premier ;
- différents types de requête pour interroger l'index : chaînes de caractères, booléen, astérisque, proximité, intervalles et bien d'autres ;
- recherche sur un champ spécifique (par exemple : titre, auteur, contenus…).

Ce chapitre n'est pas un tutoriel d'utilisation de la librairie Zend Lucene, mais seulement un exemple d'intégration de celle-ci dans l'application Jobeet, ou plus généralement, un exemple d'intégration de composants tierces parties à l'intérieur d'un projet Symfony. Pour en savoir plus au sujet de cette technologie, la documentation de Zend Lucene est disponible sur le site officiel du Zend Framework. La librairie Zend Lucene a déjà été installée dans Jobeet au chapitre précédent. Elle se trouve dans le répertoire `Search/` du framework Zend.

▸ http://framework.zend.com/

Indexer le contenu de Jobeet

Le moteur de recherche de Jobeet doit être capable de retourner toutes les offres qui correspondent aux mots-clés saisis par l'utilisateur. Cependant, avant de pouvoir rechercher une quelconque information, il est nécessaire qu'un index des annonces soit construit en amont. Ce dernier sera stocké sur le système de fichier local dans le répertoire `data/` du projet.

Créer et récupérer le fichier de l'index

Zend Lucene fournit deux méthodes pour retrouver un index selon qu'il existe déjà ou pas. Dans un premier temps, il convient de créer une méthode « helper » dans la classe `JobeetJobTable` qui retourne un index existant ou bien en génère un nouveau à la volée.

Déclaration des méthodes de création et de récupération de l'index des offres d'emploi dans le fichier 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';
}
```

Mettre à jour l'index à la sérialisation d'une offre

Chaque fois qu'une offre est créée, éditée ou bien supprimée, l'index de celle-ci doit être mis à jour afin de rester cohérent. Il est de surcroît primordial de ne pas indexer les offres expirées ou non publiées. Le meilleur endroit pour régénérer l'index à chaque fois qu'une offre est sérialisée en base de données est bien évidemment la méthode `save()` de l'objet. Cette dernière se redéfinie de la manière suivante :

Redéfinition de la méthode save() de l'objet JobeetJob dans le fichier lib/model/doctrine/JobeetJob.class.php

```
public function save(Doctrine_Connection $conn = null)
{
  // ...

  $ret = parent::save($conn);

  $this->updateLuceneIndex();

  return $ret;
}
```

La nouvelle méthode `updateLuceneIndex()` de cette même classe a pour rôle de générer l'index Lucene de cette offre au moment où elle est sérialisée dans la base de données. Le code ci-dessous illustre toute l'implémentation de cette méthode.

Implémentation de la méthode updateLuceneIndex() dans le fichier lib/model/doctrine/JobeetJob.class.php

```
public function updateLuceneIndex()
{
  $index = $this->getTable()->getLuceneIndex();
```

```
  // remove an existing entry
  if ($hit = $index->find('pk:'.$this->getId()))
  {
    $index->delete($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 URL to identify it in the search
results
  $doc->addField(Zend_Search_Lucene_Field::UnIndexed('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();
}
```

La création de l'index se déroule en trois étapes successives :

1. la première consiste à supprimer l'index existant de l'offre si ce dernier existe déjà car Zend Lucene est incapable de mettre à jour un index existant ;
2. puis le code teste si l'offre en cours est expirée ou bien si elle n'est pas encore publiée. Si c'est le cas, il est aucunement nécessaire d'indexer les données afin de ne pas risquer de les ressortir lors d'une future recherche ;
3. ensuite, un nouvel objet `Zend_Search_Lucene_Document` est créé. La clé primaire de l'offre est ajoutée à l'index mais n'est pas indexée afin qu'elle serve de référence de l'offre pour plus tard au moment de la recherche. En revanche, les valeurs des champs `position`, `company`, `location` et `description` de l'objet sont indexées mais ne sont pas stockées dans l'index : ce seront les objets réels qui seront utilisés

pour afficher les résultats. Pour finir, le document est ajouté à l'index et ce dernier est validé à l'appel de la méthode `commit()`.

Sécuriser la sérialisation d'une offre à l'aide d'une transaction Doctrine

Que se passe-t-il si l'indexation d'une offre échoue ou bien si celle-ci n'est pas correctement sauvegardée dans la base de données ? Doctrine comme Zend Lucene lancent une erreur si ce genre de cas venait à se produire. Cependant, dans certaines circonstances, l'offre peut avoir été correctement sauvegardée en base de données bien que son index correspondant n'ait pas été recréé. Afin d'éviter cela, la meilleure solution est d'encapsuler les deux procédures de mise à jour de l'objet à l'intérieur d'une transaction, et de l'annuler (« rollback ») en cas d'interception d'une exception. La transaction permet ainsi de garantir l'intégrité de l'objet. En effet, si une seule manœuvre échoue, c'est tout le processus de mis à jour de l'objet qui est interrompu et réinitialisé à son état précédent.

Implémentation d'une transaction Doctrine pour assurer la cohérence entre l'index et la base de données dans le fichier lib/model/doctrine/JobeetJob.class.php

```
public function save(Doctrine_Connection $conn = null)
{
  // ...

  $conn = $conn ? $conn : $this->getTable()->getConnection();
  $conn->beginTransaction();
  try
  {
    $ret = parent::save($conn);

    $this->updateLuceneIndex();

    $conn->commit();

    return $ret;
  }
  catch (Exception $e)
  {
    $conn->rollBack();
    throw $e;
  }
}
```

Effacer l'index lors de la suppression d'une offre

Lorsqu'une offre est supprimée du site Internet (manuellement ou via la tâche automatique de nettoyage des offres périmées), son index doit lui aussi être retiré du système de fichier afin de garantir la cohérence des informations de Jobeet. Pour automatiser cette opération, la manière de procéder qui semble la plus pertinente est bien évidemment de surcharger l'implémentation de la méthode `delete()` de la classe `JobeetJob` comme le montre le code ci-dessous.

Redéfinition de la méthode delete() des objets JobeetJob dans la classe lib/model/doctrine/JobeetJob.class.php

```
public function delete(Doctrine_Connection $conn = null)
{
  $index = $this->getTable()->getLuceneIndex();

  if ($hit = $index->find('pk:'.$this->getId()))
  {
    $index->delete($hit->id);
  }

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

Manipuler l'index des offres d'emploi

Régénérer tout l'index des offres d'emploi

Maintenant que la génération et la suppression de l'index est en place, il ne reste plus qu'à recharger les données initiales de Jobeet afin de reconstruire l'index de chaque objet. Pour ce faire, il suffit d'exécuter la tâche automatique `doctrine:data-load` utilisée à plusieurs reprises au cours de cet ouvrage.

```
$ php symfony doctrine:data-load --env=dev
```

La commande est exécutée avec l'option `--env=dev` dans la mesure où l'index est dépendant de l'environnement et que l'environnement par défaut de la ligne de commande est `cli`.

Implémenter la recherche d'informations pour Jobeet

La prochaine étape consiste à mettre en place la route, la logique métier et les vues qui rendent possible la recherche d'offres d'emploi à l'utilisateur. L'implémentation d'une telle fonctionnalité n'est qu'une question de quelques minutes. Pour commencer, il faut bien évidemment déclarer une nouvelle route dans le fichier de configuration `routing.yml` de l'application frontend.

ENVIRONNEMENTS UNIX
Accès en écriture sur l'index

Les utilisateurs de systèmes d'exploitation basés sur Unix doivent s'assurer que le fichier de l'index est accessible en écriture à la fois pour l'environnement d'exécution en ligne de commande et pour celui pour le web. Il convient donc de changer les droits du répertoire `data/` et de vérifier que l'utilisateur du serveur web et de la ligne de commande ont tous deux accès à ce répertoire en écriture.

CONFIGURATION PHP **Support des fichiers ZIP**

Des avertissements PHP au niveau de la classe `ZipArchive` peuvent être levés, ce qui signifie alors que l'extension `zip` n'est pas compilée et installée avec PHP. Ceci est un bogue connu de la classe `Zend_Loader` du framework Zend.

Ajout de la route job_search au fichier apps/frontend/config/routing.yml

```
job_search:
  url:   /search
  param: { module: job, action: search }
```

À la suite de cela, il convient d'implémenter le contrôleur `search` du module `job` comme le montre le code ci-dessous.

Implémentation de la méthode executeSearch() dans le fichier apps/frontend/modules/job/actions/actions.class.php

```
class jobActions extends sfActions
{
  public function executeSearch(sfWebRequest $request)
  {
    if (!$query = $request->getParameter('query'))
    {
      return $this->forward('job', 'index');
    }

    $this->jobs = Doctrine::getTable('JobeetJob')
      ->getForLuceneQuery($query);
  }

  // ...
}
```

Le corps de la méthode `executeSearch()` est très basique puisqu'il ne réalise que deux actions principales. Tout d'abord, une condition vérifie si l'objet correspondant à la requête contient un paramètre `query` et stocke sa valeur dans la variable `$query`. Si ce paramètre n'existe pas ou bien si sa valeur est nulle, alors la requête est transmise à l'action `index` du même module ; dans le cas contraire, c'est la nouvelle méthode `getForLuceneQuery()` qui prend en charge la récupération des offres correspondantes aux mots-clés passés en paramètre. L'implémentation de cette dernière se trouve juste après.

Implémentation de la méthode getForLuceneIndex() dans le fichier lib/model/doctrine/JobeetJobTable.class.php

```
public function getForLuceneQuery($query)
{
  $hits = $this->getLuceneIndex()->find($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);

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

La compréhension du code de cette méthode ne pose pas de réelle difficulté. La première instruction fait appel à la méthode `find()` de l'objet `Zend_Search_Lucene` représentant l'index du site Internet. La méthode `find()` retourne une collection d'objets qui répondent aux mots-clés de l'utilisateur dans l'index. Puis, la structure conditionnelle `foreach()` parcourt cette collection afin d'en récupérer les valeurs des clés primaires des offres d'emploi enregistrées dans l'index. Une requête Doctrine est enfin construite et exécutée dans le but de retourner une collection des vingt premiers objets `JobeetJob` valides qui satisfont la recherche de l'utilisateur.

Quant au template `searchSuccess.php`, il ne tient qu'en quelques lignes de PHP grâce à tous les remaniements et factorisations de code réalisés jusqu'à présent.

Contenu du template apps/frontend/modules/job/templates/searchSuccess.php

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

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

Enfin, il ne reste plus qu'à ajouter le moteur de recherche sur chaque page du site afin que tout soit définitivement opérationnel. Pour ce faire, il suffit d'ajouter le code HTML d'un formulaire basique dans le `layout` de l'application `frontend` comme le montre le code ci-après.

Intégration du moteur de recherche dans le fichier 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>
```

REMARQUE Zend Lucene : requêtes supportées pour interroger l'index

La librairie Zend Lucene définit un langage de requête très riche qui supporte notamment les opérations booléennes, joker, de proximité et bien d'autres encore. L'ensemble de ces fonctionnalités ainsi que l'utilisation de l'API sont largement documentés dans le manuel d'utilisation de Zend Lucene sur le site officiel du framework Zend à l'adresse http://framework.zend.com/manual/en/zend.search.lucene.html

Tester la méthode getForLuceneQuery() de JobeetJob

Afin de valider la bonne intégration de Zend Lucene dans Jobeet ainsi que le fonctionnement général du moteur de recherche, il convient d'écrire quelques séries de tests fonctionnels. Mais quels genres de tests unitaires est-il possible et judicieux de mettre en œuvre ? Bien évidemment, il est inutile de tester la libraire Zend Lucene elle-même dans la mesure où cela a déjà été réalisé par l'équipe de développement du Zend Framework. En fait, il s'agit de vérifier l'intégration de celle-ci au sein de la classe `JobeetJob`.

L'idéal est donc de tester la méthode `getForLuceneQuery()` qui fait le pont entre la libraire `Zend_Search_Lucene` et le moteur de recherche de Jobeet. Le code ci-dessous présente les suites de tests unitaires de la méthode `getForLuceneQuery()` à ajouter au bas du fichier `JobeetJobTest.php` en n'oubliant pas de mettre à jour le compteur de tests planifiés à la valeur 7.

Intégration des tests unitaires du moteur de recherche dans le fichier test/lib/model/JobeetJobTest.php

```
$t->comment('->getForLuceneQuery()');
$job = create_job(array('position' => 'foobar', 'is_activated'
=> false));
$job->save();
$jobs = Doctrine::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::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::getTable('JobeetJob')
  ->getForLuceneQuery('position:foobar');
$t->is(count($jobs), 0, '::getForLuceneQuery() does not return
deleted jobs');
```

L'objectif de ces trois séries de tests unitaires est de contrôler qu'une offre d'emploi non publiée ou bien supprimée n'apparaît pas dans les

résultats de recherche, alors qu'inversement, les offres qui correspondent aux critères de recherche apparaissent dans ces résultats.

Nettoyer régulièrement l'index des offres périmées

Avec le temps, il arrive que certaines offres atteignent leur date d'expiration et ne soient pas renouvelées pour une nouvelle période de trente jours. Leurs informations restent alors conservées dans l'index puisque ces offres sont toujours présentes dans la base de données. De toute évidence, il semble pertinent de nettoyer l'index régulièrement afin de supprimer toute information relative à une offre périmée. Cela aura pour effet immédiat de libérer de la place dans l'index, ce qui le rendra plus performant par la même occasion.

La manière idéale et efficace pour nettoyer l'index périodiquement est de créer et d'exécuter une commande Symfony à l'aide d'une tâche planifiée. Or, l'application Jobeet dispose depuis le chapitre 11 d'une tâche automatique de nettoyage des offres périmées de la base de données. L'objectif est donc de profiter de l'existence de cette tâche pour l'enrichir en lui intégrant le processus de mise à jour de l'index.

Implémentation du nettoyage de l'index dans la tâche JobeetJobCleanup du fichier lib/task/JobeetCleanupTask.class.php

```
protected function execute($arguments = array(), $options =
array())
{
  $databaseManager = new sfDatabaseManager($this
                                          ->configuration);

  // cleanup Lucene index
  $index = Doctrine::getTable('JobeetJob')->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()))
    {
      $hit->delete();
    }
  }
```

```
  $index->optimize();

  $this->logSection('lucene', 'Cleaned up and optimized the job
index');

  // Remove stale jobs
  $nb = Doctrine::getTable('JobeetJob')
    ->cleanup($options['days']);

  $this->logSection('doctrine', sprintf('Removed %d stale
jobs', $nb));
}
```

Le code mis en avant ici correspond à la logique nécessaire pour retirer de l'index les informations obsolètes. Pour commencer, la requête Doctrine récupère la collection d'objets `JobeetJob` ayant déjà expiré. Puis, pour chaque offre d'emploi, la méthode `find()` de l'index tente de retrouver les informations indexées et référencées à la valeur de la clé primaire de l'annonce, et stocke l'objet résultant dans la variable `$hit` en cas de succès. Dès lors, l'appel à la méthode `delete()` sur cette dernière suffit à effacer les données indexées de l'offre courante dans l'index. Enfin, la méthode `optimize()` se charge, comme son nom l'indique, d'optimiser tout l'index lorsque l'ensemble des données obsolètes a été éradiqué.

En résumé…

Ce seizième chapitre a été l'occasion d'implémenter un moteur de recherche complètement fonctionnel en moins d'une heure. Il faut toujours avoir à l'esprit de vérifier s'il existe déjà ou non des solutions aux problèmes que l'on cherche à résoudre sur un nouveau projet. Il s'agit d'abord de vérifier si la solution technique n'est pas déjà intégrée nativement dans le framework. Puis, si ce n'est pas le cas, le bon réflexe à adopter consiste à parcourir le dépôt des plug-ins de Symfony, à l'affût d'un éventuel plug-in existant. Enfin, si la solution technique ne se trouve pas en ces lieux, il ne faut pas hésiter à aller la chercher dans des ressources en ligne comme les bibliothèques Open Source telles que le Zend Framework ou les composants ezComponents.

Le chapitre suivant aborde les notions d'Ajax et de JavaScript non intrusif afin d'améliorer la réactivité du moteur de recherche en mettant à jour les résultats en temps réel à chaque fois que l'utilisateur saisit de nouvelles lettres sur son clavier dans le formulaire de recherche…

chapitre 17

Dynamiser l'interface utilisateur avec Ajax

MOTS-CLÉS :

- JavaScript, Ajax et jQuery
- Tests fonctionnels
- Couche contrôleur

Avec l'arrivée des navigateurs web modernes il y a quelques années, de nombreuses applications web se sont dotées d'interfaces riches (RIA) et dynamiques reposant sur la technologie JavaScript. L'essor des frameworks JavaScript tels que Prototype, jQuery ou MooTools, ont par la même occasion largement participé au développement de ces interfaces et plus particulièrement à l'adoption de l'Ajax.

Symfony supporte nativement les requêtes HTTP asynchrones (Ajax) et en simplifie grandement la gestion.

BONNE PRATIQUE **JavaScript non intrusif**

Une bonne pratique du développement JavaScript consiste à systématiquement externaliser le code source dans des fichiers, et ne pas l'inscrire en le mélangeant au contenu HTML. Cette solution présente des avantages certains : en matière d'organisation, il sera plus simple de s'y retrouver. Également, lorsque le JavaScript ne sera pas activé sur le poste de l'utilisateur, le code JavaScript ne sera pas chargé inutilement en mémoire. Certes, cette solution implique une requête HTTP supplémentaire, mais c'est là un moindre mal sachant que ce fichier JavaScript externe sera de toutefaçon mis en cache par le navigateur.

Le chapitre précédent a été l'occasion d'installer un moteur de recherche puissant et fonctionnel pour Jobeet à l'aide de la librairie Zend Lucene du framework Zend. L'objectif de ces prochaines pages est d'améliorer la réactivité de ce moteur de recherche en tirant partie des avantages d'Ajax afin de permettre la recherche en temps réel. Il s'agit d'une fonctionnalité que l'on retrouve par exemple sur la page d'accueil de Google.

Le formulaire de recherche est prévu pour fonctionner avec ou sans JavaScript activé. Pour cette raison, la fonctionnalité de recherche en temps réel sera alors implémentée grâce à du JavaScript non intrusif basé sur un framework JavaScript. L'utilisation de JavaScript non intrusif permet également une meilleure séparation des métiers côté client entre le HTML, la CSS et les comportements JavaScript.

Choisir un framework JavaScript

Découvrir la librairie jQuery

Au lieu de réinventer la roue et de perdre du temps à gérer les différences d'interprétation des moteurs d'exécution des navigateurs web, le développement de la recherche en temps réel s'appuiera sur l'usage du framework JavaScript jQuery. Le framework Symfony peut bien sûr fonctionner avec n'importe quelle librairie JavaScript.

jQuery est un framework JavaScript libre développé par John Resig, développeur à la fondation Mozilla, et apparu publiquement dans le courant du mois de janvier 2006. jQuery apporte une API qui simplifie la syntaxe native du langage JavaScript en fournissant une syntaxe fluide et très verbeuse ainsi que de nombreuses fonctionnalités de base. Parmi elles figurent :

- le support du parcours et de modification du DOM (Document Object Model) ;
- le support des CSS 1, CSS 2 et CSS 3 ainsi qu'une partie de XPath ;
- la gestion des événements ;
- le support de comportements comme le *drag and drop*, classement, redimensionnement… ;
- l'intégration d'effets et d'animations tels que les *zooms*, *fades in*, *fades out*… ;
- la manipulation des feuilles de style en cascade (ajout et suppression de classes, d'attributs…) ;
- le support des requêtes asynchrones (composant Ajax) ;

- l'intégration de plug-ins tierces parties ;
- la compatibilité entre les différents navigateurs…

Le projet jQuery a l'avantage de bénéficier d'une documentation abondante et très claire dotée de nombreux exemples pratiques prêts à être mis en œuvre. D'autre part, cette librairie est supportée par une communauté toujours croissante de développeurs qui participent à l'amélioration du projet en proposant librement des portions de code, des applications complètes et fonctionnelles, des retours d'expérience, ou bien encore de la documentation. Il existe également une liste de diffusion extrêmement active à partir de laquelle les utilisateurs ont la capacité de prévenir l'équipe de développement de bogues éventuels.

Télécharger et installer jQuery

Récupérer la dernière version stable du projet

L'installation de jQuery est particulièrement facile puisqu'il s'agit de télécharger un seul fichier JavaScript contenant tout le code de la librairie, puis de le charger dans les pages web à l'aide d'une balise `<script>`. La dernière version du framework jQuery est disponible en téléchargement libre sur le site officiel du projet à l'adresse http://docs.jquery.com/Downloading_jQuery#Download_jQuery. Pour commencer, le fichier source `.js` de la toute dernière version stable de jQuery doit être récupéré, puis déposé dans le répertoire `web/js` du projet Symfony.

Deux versions de la librairie sont disponibles : minifiée et non compressée. La version non compressée correspond au code source tel qu'il a été écrit par John Resig, c'est-à-dire avec les espaces, les indentations, les commentaires, les retours à ligne… Le poids du fichier final est donc largement optimisable puisqu'il contient majoritairement des caractères inutiles à son interprétation par le navigateur. En revanche, le fichier minifié est le code source final après que tous ces caractères superflus ont été supprimés, rendant alors ce dernier complètement illisible par l'humain mais beaucoup plus léger et rapide à charger. Étant donné que l'implémentation interne du code de jQuery ne fait pas l'objet de cet ouvrage, mais surtout qu'elle doit vérifier des conditions évidentes de performance, c'est la version minifiée de jQuery qui sera utilisée dans ce projet.

Charger la librairie jQuery sur chaque page du site

Le moteur de recherche d'offres d'emploi de Jobeet est présent sur chaque page du site Internet, cela implique que la librairie jQuery soit importée sur chacune d'entre elles. L'endroit idéal pour faire appel à ce nouveau JavaScript sur chaque page est, bien entendu, le layout puisqu'il

est partagé par toutes les pages du site. L'inclusion du fichier `.js` se réalise à l'aide du helper `use_javascript()` dans la section `<head>` du layout. Il faut cependant faire attention à insérer la fonction `use_javascript()` avant l'appel au helper `include_javascripts()`.

Chargement de la librairie jQuery dans le fichier apps/frontend/templates/layout.php

```
  <?php use_javascript('jquery-1.2.6.min.js') ?>
  <?php include_javascripts() ?>
</head>
```

Il est bien sûr possible d'inclure le code source de jQuery directement en écrivant la balise `<script>` manuellement mais l'utilisation de la fonction `use_javascript()` a l'avantage de s'assurer que le fichier JavaScript n'est inclus qu'une seule fois.

ASTUCE **Charger les JavaScripts à la fin de la page**

Le navigateur web interprète une page web de haut en bas, c'est-à-dire qu'il charge en premier lieu toutes les ressources qui se trouvent dans la section `<head>` de cette dernière. Les fichiers JavaScript ont le désagréable défaut d'empêcher la parallélisation des téléchargements des ressources lorsque le navigateur est en train de les récupérer. Cela a pour effet immédiat d'augmenter le temps de chargement de la page en cours et de diminuer l'accès à l'information.
Une bonne pratique consiste donc à charger les fichiers JavaScript uniquement lorsque toute la page est chargée afin de permettre à l'utilisateur d'accéder directement à l'information sans ralentissement. Pour ce faire, il suffit de déplacer l'appel au helper `include_javascripts()` juste avant la fermeture de la balise `</body>` du layout.
Les articles de la zone des développeurs de Yahoo! expliquent et donnent de nombreuses astuces pour vous aider à optimiser les performances d'une application web côté client et côté serveur :
http://developer.yahoo.com/performance/rules.html#js_bottom

Découvrir les comportements JavaScript avec jQuery

Implémenter une recherche en temps réel implique d'interroger le serveur web à chaque fois que l'utilisateur saisit une nouvelle lettre dans la boîte de recherche. Cette requête au serveur est ensuite suivie par une réponse, dans un format donné (HTML, JSON, XML…), et interceptée par le navigateur afin de mettre à jour certaines régions de la page web sans avoir à en rafraîchir l'intégralité.

Intercepter la valeur saisie par l'utilisateur dans le moteur de recherche

Traditionnellement, les comportements JavaScript se définissent et s'exécutent grâce aux attributs HTML `on*` tels que `onsubmit`, `onfocus`, `onblur`, `onclick`… Cependant, cette pratique désormais obsolète a le principal défaut d'engendrer des erreurs si le JavaScript n'est pas activé sur le navigateur, mais va également à l'encontre du principe de non-intrusivité expliqué plus haut. De plus, elle lie fortement le code JavaScript au document HTML. L'approche avec jQuery est différente et plus intelligente puisqu'il s'agit d'attacher les comportements aux événements du DOM une fois la page complètement chargée. De cette manière, si le support du JavaScript est désactivé dans le navigateur, aucun comportement ne sera déclaré, et le formulaire continuera de marcher comme avant. C'est le principe du Javascript non-intrusif.

La première étape du développement de la recherche en temps réel consiste à intercepter la valeur de la boîte de saisie à chaque fois que l'utilisateur tape une nouvelle lettre. Avec jQuery, cela tient en quelques lignes seulement grâce à son API fluide.

```
$('#search_keywords').keyup(function(key)
{
  if (this.value.length >= 3 || this.value == '')
  {
    // do something
  }
});
```

Cette syntaxe que fournit jQuery permet d'ajouter facilement un comportement à un événement particulier, ici `keyup`, à un ou plusieurs objets du DOM en spécifiant la valeur de son attribut unique `id` ou de sa classe CSS. Dans le cas présent, la chaîne `#search_keywords` indique que l'on applique le comportement `keyup` à l'objet dont l'attribut `id` (représenté par le symbole dièse #) porte la valeur `search_keywords`. La fonction spéciale `$()` de jQuery permet de récupérer un objet ou toute une collection d'objets du DOM qui correspondent à l'expression passée en paramètre. Cette fonction est en quelque sorte un équivalent amélioré des traditionnelles méthodes `getElementById()`, `getElementByTagName()`… Enfin, le code de la fonction anonyme du comportement associé à l'événement `keyup` se charge d'exécuter toute la logique métier de la recherche en temps réel. L'objet `this` représente ici l'objet DOM récupéré grâce à la fonction `$()`. Dans le cas présent, il s'agit du nœud DOM de la balise `<input/>` ; c'est pour cette raison qu'il est possible d'accéder directement à la valeur saisie par l'utilisateur grâce à `this.value`.

Exécuter un appel Ajax pour interroger le serveur web

Chaque fois que l'utilisateur tape sur une touche de son clavier, jQuery exécute la fonction anonyme définie dans le code ci-dessus à condition que la chaîne saisie soit composée de plus de 3 caractères, ou que la valeur du champ ait été réinitialisée.

Réaliser un appel Ajax au serveur consiste simplement à appliquer la méthode `load()` sur le nœud DOM dans lequel on souhaite charger la réponse renvoyée par le serveur. Grâce à jQuery, il n'y a plus besoin de déclarer soi-même l'objet `XMLHTTPRequest` (ou `ActiveX` dans le cas d'Internet Explorer) et de gérer le traitement de la réponse en fonction du code HTTP renvoyé dans les en-têtes de la réponse. Toutes ces étapes fastidieuses sont gérées automatiquement par la méthode `load()` de jQuery.

```
$('#search_keywords').keyup(function(key)
{
  if (this.value.length >= 3 || this.value == '')
  {
    $('#jobs').load(
      $(this).parents('form').attr('action'), { query:
this.value + '*' }
    );
  }
});
```

La méthode `load()` accepte trois arguments dont seul le premier est obligatoire. Cet argument correspond à l'URL à appeler, et dans le cas présent, il s'agit de la valeur de l'attribut `action` de la balise `<form>` du moteur de recherche. Le second argument est un tableau associatif de paramètres à envoyer au serveur tandis que le troisième est le nom d'une fonction JavaScript à exécuter lorsque la requête Ajax sera complète et que le serveur aura renvoyé une réponse. Dans le cadre de Jobeet, ce dernier argument n'est pas nécessaire et est donc omis. En revanche, la méthode `load()` du script ci-dessus accueille un objet anonyme contenant la propriété `query` à transmettre au serveur. Cette dernière contient la valeur saisie par l'utilisateur, suffixée par le symbole étoile *.

Récupérer dynamiquement l'URL du formulaire en parcourant l'arbre DOM du document apporte de sérieux avantages. Le premier est bien évidemment qu'il n'est pas nécessaire de recopier ou de régénérer soi-même l'URL à l'aide du helper `url_for()`. D'autre part, cela rend le code JavaScript indépendant de son implémentation dans l'application. Il sera ainsi plus facile de réutiliser ce code dans un autre projet sans avoir à le modifier. Enfin, comme l'URL utilisée est finalement la même que celle du moteur de recherche en mode « dégradé » (i.e. sans JavaScript), le développement côté serveur s'en voit donc allégé.

Cacher dynamiquement le bouton d'envoi du formulaire

Lorsque JavaScript est activé sur le navigateur, le bouton d'envoi du formulaire n'a plus véritablement d'utilité puisque toutes les actions relatives à la recherche sont gérées dynamiquement par les comportements JavaScript. De ce fait, il convient de masquer ce bouton de la page en appelant la méthode jQuery `hide()` sur l'objet DOM de ce dernier. La méthode `hide()` modifie les valeurs des propriétés CSS de l'objet afin de le cacher.

```
$('.search input[type="submit"]').hide();
```

Cette instruction montre également combien il est aisé d'accéder à un ou plusieurs objets du DOM en fournissant une expression plus ou moins complexe à la fonction `$()`. Ici, le bouton d'envoi du formulaire est ciblé par l'expression CSS 3 indiquant qu'il se trouve dans un conteneur parent ayant un attribut `class` dont la valeur est `search`.

Informer l'utilisateur de l'exécution de la requête Ajax

À chaque fois qu'un appel Ajax est exécuté, la région de la page à modifier n'est pas mise à jour immédiatement. Le navigateur doit en effet attendre que la réponse du serveur lui revienne avant de mettre à jour le document. Le temps d'attente de la réponse est variable en fonction de la complexité de l'action exécutée, de l'engorgement des réseaux, ou bien encore de la qualité de la bande passante de l'utilisateur. Il s'avère alors judicieux d'informer celui-ci que son action est en cours de traitement.

Faire patienter l'utilisateur avec un « loader »

L'une des manières les plus traditionnelles d'y parvenir est d'afficher une petite animation chargée par le navigateur au chargement de la page, à l'origine cachée grâce à une propriété CSS adéquate.

Formulaire de recherche à remplacer dans le fichier 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" />
```

ASTUCE Créer son propre loader

Le loader par défaut est optimisé pour le layout de Jobeet. Le site Internet suivant http://www.ajaxload.info propose aux développeurs un moyen simple de générer leurs propres images de chargement à partir d'une large banque de loaders prédéfinis. Un formulaire de génération invite l'utilisateur à déterminer l'image de chargement à utiliser ainsi que la couleur des avant et arrière plans à partir de leur code hexadécimal respectif.

```
    <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>
```

Déplacer le code JavaScript dans un fichier externe

Pour l'instant, le code JavaScript implémenté pour faire fonctionner le formulaire de recherche ne dispose pas encore d'une place dédiée dans le projet. L'idéal est donc de déplacer ce code JavaScript autonome dans un fichier `search.js`, sauvegardé dans le répertoire `web/js/`.

Contenu du fichier 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(); }
      );
    }
  });
});
```

La méthode jQuery `load()` définit à présent une nouvelle fonction anonyme de rappel (*callback*) en troisième argument. Cette nouvelle fonction est appelée automatiquement lorsque la requête Ajax est complètement terminée. Elle a donc en charge de masquer à nouveau l'image de chargement lorsque la page a bien été mise à jour.

Maintenant que le fichier `search.js` est créé, il ne reste plus qu'à l'appeler dans le layout afin qu'il soit chargé par le navigateur pour chaque page du site.

Appel du fichier search.js dans le fichier apps/frontend/templates/layout.php

```
<?php use_javascript('search.js') ?>
```

REMARQUE JavaScript comme une action

Bien que le code JavaScript exécuté jusqu'à présent pour le moteur de recherche soit statique, il arrive parfois d'avoir besoin d'appeler du code PHP (par exemple pour utiliser le helper `url_for()`).

JavaScript n'est en fait qu'un format de plus comme le HTML, et comme on l'a vu dans les chapitres précédents, le framework Symfony permet la gestion des différents formats de sortie avec une grande simplicité. Dans la mesure où le fichier JavaScript contiendra le comportement pour une page, il est possible d'obtenir la même URL pour le fichier JavaScript à ceci près qu'elle se terminera par l'extension .js. Par exemple, si l'on souhaite créer un fichier pour le moteur de recherche, il suffit d'éditer la route `job_search` comme ci-dessous, puis de créer le template associé `searchSuccess.js.php`.

```
job_search:
  url:   /search.:sf_format
    param: { module: job, action:
search, sf_format: html }
  requirements:
    sf_format: (?:html|js)
```

Manipuler les requêtes Ajax dans les actions

Si JavaScript est activé, jQuery interceptera les touches appuyées par l'utilisateur dans le formulaire de recherche, puis appellera ensuite l'action `search` du module `job`. Cette même action sera aussi exécutée si JavaScript est désactivé lorsque l'utilisateur soumettra le formulaire en pressant la touche *Entrée* de son clavier, ou bien en cliquant le bouton *search* visible à l'écran.

Déterminer que l'action provient d'un appel Ajax

Quel que soit l'état du support du JavaScript dans le navigateur, l'action `search` sera toujours exécutée pour le formulaire de recherche. Il est alors naturel de se demander de quelle manière il est possible de différencier une requête HTTP classique d'une requête en provenance d'un appel Ajax.

La réponse se trouve en réalité dans les en-têtes envoyés au serveur. En effet, un appel Ajax transmet sa requête au serveur avec l'en-tête `X-Requested-With` qui porte la valeur `XMLHttpRequest`. Cette information est en revanche absente des en-têtes pour une requête HTTP traditionnelle. L'objet `sfWebRequest` du framework Symfony intègre une méthode `isXmlHttpRequest()` qui retourne la valeur `true` si la requête appelée provient d'un appel Ajax.

Implémentation de la méthode isXmlHttpRequest() dans l'action search du fichier apps/frontend/modules/job/actions/actions.class.php

```
public function executeSearch(sfWebRequest $request)
{
  if (!$query = $request->getParameter('query'))
  {
    return $this->forward('job', 'index');
  }

  $this->jobs = Doctrine::getTable('JobeetJob')
                  ->getForLuceneQuery($query);

  if ($request->isXmlHttpRequest())
  {
    return $this->renderPartial('job/list', array('jobs' =>
$this->jobs));
  }
}
```

Dans la mesure où jQuery ne rechargera pas la page entière mais remplacera uniquement le contenu de l'élément DOM `#jobs` par le contenu HTML de la réponse, la décoration de la page par le layout doit être désactivée. Retirer le layout dans le cadre d'une requête Ajax est un

PRATIQUE isXmlHttpRequest() et les frameworks JavaScript

La méthode `isXmlHttpRequest()` de l'objet `sfWebRequest` de Symfony fonctionne avec la majorité des librairies JavaScript actuelles telles que Prototype, Mootools, Dojo et jQuery bien sûr.

besoin fréquent, c'est pour cette raison que c'est le comportement adopté par défaut dans Symfony.

De plus, au lieu de retourner l'intégralité du template, on se contente uniquement de renvoyer le contenu du template partiel `job/list`. La méthode `renderPartial()` employée dans l'action retourne le partiel évalué en guise de réponse au lieu du template `searchSuccess.php` complet.

Message spécifique pour une recherche sans résultat

Pour l'instant, la requête Ajax retourne toujours le même template, qu'il y ait des résultats ou non. Le scénario idéal consiste donc à informer l'utilisateur si aucun résultat ne correspond à sa recherche en lui affichant un court message à la place d'une page blanche. La méthode `renderText()` permet de réaliser ce type d'opération sans effort comme l'explique le code ci-dessous.

Implémentation de la méthode renderText() dans l'action search du fichier apps/frontend/modules/job/actions/actions.class.php

```
public function executeSearch(sfWebRequest $request)
{
  if (!$query = $request->getParameter('query'))
  {
    return $this->forward('job', 'index');
  }

  $this->jobs = Doctrine::getTable('JobeetJob')
                    ->getForLuceneQuery($query);

  if ($request->isXmlHttpRequest())
  {
    if ('*' == $query || !$this->jobs)
    {
      return $this->renderText('No results.');
    }
    else
    {
      return $this->renderPartial('job/list',
                                  array('jobs' => $this->jobs));
    }
  }
}
```

ASTUCE Retourner un composant

De la même manière qu'avec la méthode `renderPartial()`, il existe une méthode `renderComponent()` qui permet de retourner le contenu généré à la suite de l'exécution d'un composant. Cette méthode accepte trois arguments dont les deux premiers sont obligatoires. Le premier paramètre correspond au nom du module dans lequel se situe le composant alors que le second est le nom du composant à exécuter. Enfin, le troisième argument facultatif est un tableau associatif de variables à transmettre au composant.

Simuler une requête Ajax avec les tests fonctionnels

Comme le navigateur interne de Symfony est incapable d'exécuter ou de simuler du code JavaScript, les tests fonctionnels de la recherche en temps réel sont alors impossibles. Néanmoins, il reste toujours la possibilité de tester les appels Ajax en fournissant manuellement l'URL à atteindre avec ses paramètres ainsi que l'en-tête `X_REQUESTED_WITH` qu'envoient tous les frameworks JavaScript comme jQuery avec sa requête Ajax. Le template partiel des offres d'emploi rendu en guise de réponse à une requête Ajax peut ainsi être testé à l'aide du testeur `response`.

Scénario de tests fonctionnels à ajouter au bas du fichier 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', 3)->
  end()
;
```

La méthode `setHttpHeader()` fixe un en-tête HTTP pour la toute prochaine requête appelée avec le navigateur interne de Symfony.

En résumé…

Le chapitre précédent a montré pas à pas comment implémenter un moteur de recherche entièrement fonctionnel grâce à la technologie Zend Lucene. Ce chapitre a de son côté permis d'améliorer l'expérience utilisateur en rendant le moteur plus réactif grâce à l'implémentation d'une recherche en temps réel mise en place grâce à jQuery.

Le framework Symfony fournit non seulement tous les outils essentiels pour construire des applications MVC en toute simplicité, mais s'accorde en plus très bien avec de nombreux autres composants. Comme toujours, il convient d'avoir recours aux meilleurs outils pour chaque tâche dédiée, et c'est exactement comme ça que se comportent Zend Lucene pour la recherche et jQuery pour le JavaScript.

Le chapitre suivant aborde un nouveau point essentiel des applications web professionnelles : l'internationalisation et la régionalisation. Il sera question du support de différentes cultures de l'utilisateur, de traduction de contenus, de traduction d'offres d'emploi en base de données ou bien encore des formatages de dates ou de monnaies par exemple.

chapitre 18


Paris, France
Web Developer
Company 128
AND 22 MORE...
About Jobeet
Full feed
Jobeet API
Become an affiliate
Language
French
ok
English
French
Jobeet powered by symfony

Internationalisation et localisation

MOTS-CLÉS :

- Interface multilingue
- Traduction de contenus
- Formatage des dates et monnaies

Nombreux sont les sites Internet qui disposent d'une interface multilingue pour leurs utilisateurs. La traduction d'une application web en différentes langues n'est pas une tâche aisée pour les développeurs car elle implique de faire face à de nombreuses contraintes. Parmi elles figurent bien évidemment les traductions des contenus statiques et dynamiques (base de données, pluriels), le sens de lecture de la langue ou bien encore le formatage des dates, heures, nombres et autres devises monétaires.

Le framework Symfony intègre de nombreux outils pour répondre à tous ces besoins, explicités dans ce dix-huitième chapitre.

Le développement de l'application Jobeet est sur le point de toucher à sa fin. De nombreuses fonctionnalités ont été implémentées tout au long de cet ouvrage pour rendre l'application à la fois dynamique, fonctionnelle, confortable à prendre en main de part l'écriture de code JavaScript non-intrusif, et enfin, dans l'air du temps grâce à ses flux de syndication de contenu et ses autres services web.

Que sont l'internationalisation et la localisation ?

C'est donc le moment de s'intéresser à la mise en place d'une interface multilingue pour Jobeet afin de permettre aux utilisateurs francophones et anglophones de naviguer à travers les offres d'emploi dans la langue qui leur convient le mieux. De nombreux ajustements à travers toute l'application devront être opérés pour répondre à ce besoin, mais heureusement le framework Symfony dispose de tous les outils nécessaires pour simplifier l'intégration de l'internationalisation (I18N) et de la localisation (L10N) d'un projet. À quoi correspondent exactement ces deux concepts ? Qu'est-ce qui les différencie ?

L'internationalisation, traditionnellement abrégée I18N, concerne tout le processus de conception d'une application logicielle ayant vocation à être adaptée pour de multiples langues et régions sans qu'il y ait besoin de changements techniques.

La localisation, également nommée régionalisation et abrégée L10N, correspond au processus d'adaptation d'une application logicielle pour une région ou une langue spécifique, en intégrant des composants propres à cet élément local (par exemple : formatage des dates, heures, nombres, monnaies…) ainsi que la traduction des textes.

Comme toujours, le framework Symfony n'a pas réinventé la roue et doit son support natif de l'internationalisation et de la localisation au standard ICU.

CULTURE INFORMATIQUE
Origine des abréviations I18N et L10N

Les termes internationalisation et localisation, particulièrement contraignants à écrire, ont été respectivement abrégés en I18N et L10N. Ces abréviations n'ont pas été très difficiles à trouver, puisque I18N vient tout simplement du fait qu'il y a 18 lettres entre le *i* et le *n* du terme anglais *internationalization*. Sur le même principe, l'abréviation du terme anglais *localization* a conduit à L10N.

TECHNOLOGIE **Le standard ICU**

ICU est l'acronyme pour *International Components for Unicode*. Initialement développée et supportée par les plateformes C, C++ et Java, la bibliothèque Open Source ICU est devenue une solution portable reconnue pour le traitement de textes internationaux. Avec le temps, de nombreux portages ont ainsi pu être réalisés sur différents langages de programmation, d'après la librairie C originale. ICU fournit un certain nombre de fonctionnalités telles que la conversion des contenus en (et depuis) Unicode, la comparaison de chaînes de caractères, le formatage des dates, nombres, heures et devises ou bien encore les calculs temporels en fonction du calendrier local. Pour en savoir plus au sujet du standard ICU, nous vous invitons à consulter le site officiel à l'adresse http://site.icu-project.org/

L'utilisateur au cœur de l'internationalisation

L'utilisateur est au cœur du processus d'internationalisation de l'application, car sans lui ce concept n'a aucune raison d'être. En effet, lorsqu'un site Internet est disponible en plusieurs langues ou pour différentes régions du monde, l'utilisateur est responsable du choix de celle qui lui convient le mieux. C'est pour cette raison que dans Symfony, la notion de « culture » est propre à l'utilisateur courant pour englober tout ce qui concerne les problématiques d'internationalisation et de localisation.

Référence L'objet sfUser

L'ensemble des concepts et fonctionnalités propres à l'utilisateur comme la culture ou bien les droits d'accès ont été présentés au chapitre 13 de cet ouvrage.

Paramétrer la culture de l'utilisateur

Les fonctionnalités d'internationalisation et de localisation de Symfony reposent toutes sur la culture de l'utilisateur qui réside dans sa session persistante. La culture est en fait la combinaison de deux éléments propres à l'utilisateur : sa langue et son pays. Par exemple, la culture d'une personne qui parle le français est `fr` tandis que la culture pour une personne française qui habite en France est `fr_FR`. Les sections qui suivent présentent les différentes manières de fixer la valeur de la culture de l'utilisateur et comment agir avec elle.

Définir et récupérer la culture de l'utilisateur

L'objet `sfUser` dispose de deux méthodes très pratiques pour gérer la valeur de la culture qui lui est associée. Il s'agit des méthodes `setCulture()` et `getCulture()` qui permettent respectivement de fixer la valeur de la culture et de l'obtenir.

```
// Depuis les actions
$this->getUser()->setCulture('fr_BE');
echo $this->getUser()->getCulture();

// Depuis les templates
$sf_user-> setCulture('fr_BE');
echo $sf_user->getCulture();
```

Convention Formatage de la culture

La valeur de la culture est un format normalisé. En effet, la langue est codée sur deux caractères minuscules comme le spécifie le standard ISO 639-1 alors que le pays est codé avec deux caractères en majuscules, d'après les spécifications du standard ISO 3166-1.

Modifier la culture par défaut de Symfony

Par défaut, la culture de l'utilisateur est automatiquement initialisée avec la valeur de la culture inscrite initialement dans le fichier de configuration `settings.yml` de l'application, à la section `default_culture`. Cette directive de configuration peut alors être redéfinie pour attribuer une nouvelle culture par défaut à tous les nouveaux utilisateurs arrivant sur l'application.

Astuce Changer de culture

Dans Symfony, la culture est gérée par l'objet User, et est notamment sauvegardée dans la session persistante de ce dernier. Au cours du développement, si la culture par défaut est amenée à être modifiée, alors il sera nécessaire de vider les cookies de session du navigateur afin que la nouvelle configuration soit prise en compte par le navigateur, et que les changements sur l'interface prennent effet immédiatement.

Modification de la valeur par défaut de la culture dans le fichier de configuration apps/frontend/config/settings.yml

```
all:
  .settings:
    default_culture: it_IT
```

Déterminer les langues favorites de l'utilisateur

Pour un site international, il n'est pas toujours judicieux de forcer la valeur de la langue par défaut pour l'utilisateur. En effet, si un utilisateur anglophone arrive sur le site et que la langue par défaut pour ce dernier est le français, il se sentira contraint de changer lui-même la langue du site. Cela a pour effet immédiat dans le meilleur des cas de lui faire recharger la page sur laquelle il se trouve, ou dans le pire de lui faire quitter définitivement le site…

L'idéal est donc de fixer la valeur de la culture par défaut d'après les informations transmises par le navigateur dans l'en-tête HTTP `Accept-Language`. En effet, cet en-tête envoie au serveur une liste prédéfinie des langues du navigateur, d'après sa configuration. Les langues par défaut du navigateur présentes dans cette liste sont triées les unes par rapport aux autres par ordre de préférence. Ainsi, si le navigateur indique au serveur qu'il est paramétré à cet instant en langue anglaise, c'est qu'il y a fort à parier que l'utilisateur qui se trouve face à l'écran parle, lit et comprend l'anglais.

Sachant cela, il paraît pertinent de s'appuyer sur cette information pour fixer automatiquement la langue par défaut de l'utilisateur. Dans Symfony, c'est l'appel aux méthodes `getLanguages()` et `getPreferredCulture()` de l'objet `sfWebRequest` qui permet de déterminer la configuration du navigateur. La méthode `getLanguages()` de l'objet de requête a pour rôle de renvoyer un tableau des langues supportées par le navigateur du client.

```
// Depuis une action
$languages = $request->getLanguages();
```

Néanmoins, la plupart du temps, l'application web ne sera pas disponible pour les 136 langues majeures de la planète ; c'est pourquoi il est préférable d'avoir recours à la méthode `getPreferredCulture()` qui se charge de deviner et de retourner la meilleure langue en comparant les langues prédéfinies du navigateur du client avec celles qui sont supportées par le site web.

```
// Depuis une action
$language = $request->getPreferredCulture(array('en', 'fr'));
```

Lors de l'exécution de cet appel à `getPreferredCulture()`, la langue renvoyée par la méthode sera l'anglais ou le français d'après la liste des langues par défaut du navigateur de l'utilisateur. Si aucune des deux ne correspond, alors c'est la première du tableau passé en paramètre qui est choisie.

CHOIX DE CONCEPTION Définir la langue d'après la configuration du navigateur

Il faut garder à l'esprit que cette technique n'est pas fiable à 100 % dans la mesure où elle n'exclut pas par exemple qu'un utilisateur étranger se trouve devant un poste qui ne lui appartient pas. Le cas le plus typique est lorsque l'utilisateur se connecte depuis l'ordinateur d'un cybercafé d'un pays étranger où il séjourne pour ses vacances.

Utiliser la culture dans les URLs

Le site Internet de Jobeet sera disponible en deux langues : anglais et français. Il est important de rappeler qu'une URL représente uniquement une seule ressource, et pour l'instant toutes les URLs de Jobeet ne sont pas adaptées pour accéder à une même ressource disponible dans deux langues différentes. Les prochaines sections expliquent comment prendre en compte la notion de culture dans les URLs et comment rediriger automatiquement l'utilisateur sur la page qui lui correspond le mieux dès son arrivée sur le site, d'après la configuration de son navigateur envoyée au serveur.

Transformer le format des URLs de Jobeet

L'ensemble des pages du site Internet de Jobeet sera accessible dans les deux langues, ce qui implique qu'il faille obligatoirement remanier le format des URLs, afin que l'utilisateur puisse distinguer clairement sur quelle version du site il se trouve. Pour résoudre cette problématique, il est nécessaire de rouvrir le fichier de configuration `routing.yml` et d'éditer toutes les routes, à l'exception des routes `api_jobs` et `homepage`, pour qu'elles embarquent la variable spéciale `sf_culture`. Les routes simples se dotent à présent du motif `/:sf_culture` au début de l'URL tandis que les collections de routes surchargent leur option de configuration respective `prefix_path` avec `/:sf_culture` en tête.

Intégration de la variable spéciale sf_culture dans les URLs du fichier 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 présente dans une route, Symfony utilise automatiquement sa valeur pour changer la culture de l'utilisateur. Néanmoins, l'objectif est de rediriger ce dernier directement sur la version traduite du site qui lui correspond au mieux d'après la configuration de son navigateur.

Attribuer dynamiquement la culture de l'utilisateur d'après la configuration de son navigateur

L'application Jobeet a besoin d'autant de pages d'accueil que de langues supportées (`/en/`, `/fr/`, etc.), c'est pourquoi la page d'accueil par défaut doit rediriger l'utilisateur vers celle qui correspond à la culture qu'il a sélectionnée. Or, à son premier passage sur le site, cette dernière n'est pas encore définie et l'utilisateur se voit automatiquement attribué la culture par défaut du fichier de configuration `settings.yml`. Dans l'idéal, il serait plus judicieux de définir sa culture dynamiquement en se basant sur la liste des langues configurées dans son navigateur et transmises dans les en-têtes de la requête HTTP.

Déclarer une route pour la page d'accueil régionalisée

Comme toujours, une ressource est rendue accessible à condition qu'une nouvelle route dédiée lui soit consacrée dans le fichier de configuration

routing.yml. Le code ci-dessous est à ajouter au fichier routing.yml. Il déclare une route localized_homepage qui fait usage de la variable spéciale sf_culture de Symfony pour déterminer dans quelle langue sera rendue la page d'accueil de Jobeet. La section requirements établit une règle pour forcer la valeur de la variable sf_culture à fr ou en.

Déclaration de la route localized_homepage dans le fichier apps/frontend/config/routing.yml

```
localized_homepage:
  url:   /:sf_culture/
  param: { module: job, action: index }
  requirements:
    sf_culture: (?:fr|en)
```

L'étape suivante consiste à implémenter la méthode isFirstRequest() suivante dans la classe myUser de l'application frontend. Cette méthode retourne une valeur booléenne indiquant si oui ou non l'utilisateur arrive pour la première fois sur le site Internet, en stockant cette dernière dans une variable de session.

Implémentation de la méthode isFirstRequest() dans la classe myUser du fichier apps/frontend/lib/myUser.class.php

```
public function isFirstRequest($boolean = null)
{
  if (is_null($boolean))
  {
    return $this->getAttribute('first_request', true);
  }
  else
  {
    $this->setAttribute('first_request', $boolean);
  }
}
```

Ceci étant fait, il ne reste plus qu'à implémenter cette méthode dans l'action index du module job pour déterminer automatiquement la culture qui correspond au mieux à l'utilisateur d'après la configuration du navigateur que ce dernier envoie au serveur par le biais des en-têtes HTTP.

Déterminer la meilleure culture pour l'utilisateur

Pour finaliser la détection automatique de la meilleure culture à attribuer à l'utilisateur à sa première arrivée sur l'application Jobeet, il est nécessaire d'implémenter la logique métier relative à ce besoin dans la méthode executeIndex() du module job. Cette dernière fait appel à la méthode isFirstRequest() définie juste avant pour déterminer s'il s'agit ou non du premier passage de l'utilisateur sur la page d'accueil de Jobeet.

Implémentation de la logique métier de détermination de la meilleure culture de l'utilisateur dans le fichier 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::getTable('JobeetCategory')-
>getWithJobs();
}
```

L'implémentation de ces quelques nouvelles lignes de PHP mérite quelques explications. La première condition vérifie que la variable `sf_culture` n'est pas présente dans la requête, ce qui signifie aussi que l'utilisateur est entré sur le site depuis l'URL racine `/` définie par la route `homepage`. Si c'est effectivement le cas et que l'utilisateur arrive pour la première fois de sa session sur le site, alors on lui affecte la culture qui correspond au mieux à la configuration de son navigateur. Dans le cas contraire, c'est sa culture courante qui est utilisée.

Enfin, l'utilisateur est redirigé automatiquement vers la page d'accueil (route `localized_homepage`) traduite dans sa langue. Il est bon de noter que la variable `sf_culture` n'est pas passée explicitement dans l'appel de la redirection puisque Symfony se charge de l'ajouter lui-même d'après la valeur de la culture fixée dans l'objet `sfUser`.

Restreindre les langues disponibles à toutes les routes

La route `localized_homepage` établit une contrainte forte dans sa section `requirements` afin de forcer la variable `sf_culture` aux deux seules valeurs `fr` et `en`. De ce fait, si un utilisateur tente d'accéder à la page d'accueil de Jobeet via l'URL `/it/`, Symfony retournera automatiquement une page d'erreur 404. Il est donc important de sécuriser l'ensemble des routes de Jobeet qui embarquent la variable `sf_culture` en leur ajoutant à chacune cette restriction dans le fichier de configuration `routing.yml`.

```
requirements:
  sf_culture: (?:fr|en)
```

Tester la culture avec des tests fonctionnels

Mettre à jour les tests fonctionnels qui échouent

Le moment est venu de s'intéresser de nouveau aux tests fonctionnels dans la mesure où une nouvelle fonctionnalité a été implémentée et que toutes les URLs ont été modifiées. La modification des URLs entraîne naturellement l'échec des tests fonctionnels, c'est pourquoi il faut d'abord les corriger avant d'en ajouter de nouveaux. La résolution des tests fonctionnels erronés nécessite seulement d'ajouter la chaîne `/en` au début de toutes les URLs des fichiers de tests du répertoire `test/functional/frontend`, y compris dans le fichier `lib/test/JobeetTestFunctional.class.php`. L'exécution de toute la suite de tests fonctionnels permet de s'assurer que tous les tests ont été correctement fixés.

```
$ php symfony test:functional frontend
```

Tester les nouvelles implémentations liées à la culture

La prise en compte de la culture dans les URLs ainsi que l'implémentation de la sélection automatique de la culture de l'utilisateur à son arrivée sur le site conduisent à écrire trois nouveaux scénarios de test, dont voici le descriptif :

- la culture de l'utilisateur est devinée automatiquement par Symfony à sa première requête ;
- les seules cultures disponibles sont `fr` et `en`, les autres provoquent des pages d'erreur 404 ;
- la culture de l'utilisateur est devinée uniquement à sa première requête.

Le testeur `user` inclut une méthode `isCulture()` qui teste la valeur de la culture courante de l'utilisateur. C'est cette méthode qui est majoritairement employée dans ces trois scénarios de tests. Le code ci-dessous définit la suite de tests de ces trois scénarios dans le fichier `JobActionsTest`.

Tests fonctionnels de la culture dans le fichier 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('/')->
  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('/')->
  isRedirected()->followRedirect()->
  with('user')->isCulture('fr')
;
```

Ici, la présence de la méthode `restart()` permet de réinitialiser le navigateur de tests, et plus particulièrement les cookies et la session de l'utilisateur qui contient la valeur de la culture. Pour s'assurer que la langue favorite de l'utilisateur est bien définie automatiquement par Symfony, l'en-tête `ACCEPT_LANGUAGE` adéquat doit être envoyé au serveur à l'aide de la méthode `setHttpHeader()` du navigateur de tests. Enfin, la méthode `isCulture()` du testeur `user` permet de contrôler la valeur de la culture attribuée à l'utilisateur par le framework lors de ses différentes requêtes au serveur.

Changer de langue manuellement

Pour l'instant la culture de l'utilisateur est définie automatiquement à son arrivée sur le site de Jobeet. Néanmoins, cette méthode d'attribution de la culture n'est pas fiable à cent pour cent dans la mesure où elle n'exclut pas qu'un utilisateur se connecte depuis un ordinateur qui n'est pas le sien, et qui par conséquent est potentiellement configuré différemment.

C'est par exemple le cas pour un utilisateur qui se connecte au site depuis un cybercafé d'un pays étranger dans lequel il séjourne temporairement. Pour cette raison, il convient de mettre en place un moyen lui permettant de changer manuellement la langue dans laquelle il souhaite consulter le site. Avec Symfony, ceci peut être réalisé tout simplement à l'aide d'un composant entièrement fonctionnel figurant dans le plug-in officiel `sfFormExtraPlugin`.

Installer le plug-in sfFormExtraPlugin

Afin que l'utilisateur puisse changer de langue manuellement, un formulaire de sélection de langue doit être ajouté dans le layout de l'application. Le framework de formulaire de Symfony ne fournit pas ce type de fonctionnalité nativement mais comme il s'agit d'un besoin particulièrement récurrent dans les sites à vocation internationale, l'équipe de développement de Symfony a développé et maintient le plug-in `sfFormExtraPlugin`.

Ce plug-in contient un certain nombre de validateurs, de widgets et d'autres formulaires qui ne peuvent être inclus par défaut dans le framework dans la mesure où ils sont trop spécifiques, ou parce qu'ils possèdent des dépendances externes avec d'autres composants tels que le framework JavaScript jQuery par exemple.

Comme toujours, l'installation d'un plug-in est automatisée avec l'aide de la commande `plugin:install` de Symfony.

```
$ php symfony plugin:install sfFormExtraPlugin
```

Comme de nouvelles classes sont livrées par le plug-in, le cache de Symfony doit être vidé afin de les prendre en compte dans le fichier d'autochargement de classes.

```
$ php symfony cc
```

ASTUCE Utiliser des widgets « riches »

Le plug-in `sfFormExtraPlugin` contient de nombreux widgets qui nécessitent l'installation de dépendances externes comme des librairies JavaScript telles que jQuery ou TinyMCE. Parmi les composants livrés par ce plug-in figurent un widget de sélection de dates en JavaScript (*date picker*), un éditeur WYSIWYG basé sur le célèbre TinyMCE, un système antispam (« captcha ») reposant sur le service ReCaptcha, une boîte de saisie d'autocomplétion à la manière de celle de Google, etc. Il suffit de lire la documentation de ces outils supplémentaires sur le site officiel de Symfony afin de découvrir bien d'autres surprises utiles.

Intégration non conforme du formulaire de changement de langue

Le plug-in `sfFormExtraPlugin` fournit la classe de formulaire `sfFormLanguage` pour gérer la sélection d'une langue parmi une liste prédéfinie d'autres langues. L'ajout de ce formulaire peut être réalisé directement dans le layout, bien que ce ne soit pas du tout la meilleure manière de procéder…

```
<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>
```

AVERTISSEMENT Contre-exemple d'installation du formulaire à ne pas reproduire !

Le code présenté ici n'a aucune vocation à être implémenté de cette manière car il ne se conforme pas à la logique du modèle de conception MVC ! Il sert uniquement en guise d'exemple pour montrer ce qu'il ne faut pas faire ni être tenté de faire par la suite. Il s'agit en effet d'une mauvaise pratique à ne pas reproduire dans de futurs développements. La suite de cette section explique comment l'implémenter proprement dans Symfony.

Intégration du formulaire de changement de langue avec un composant Symfony

Les composants Symfony à la rescousse

Le code précédent révèle un gros défaut de conception. Il s'agit de la création du formulaire qui ne se trouve pas du tout à sa place selon les principes établis par le modèle MVC. En effet, la création du formulaire n'appartient aucunement à la couche de la Vue mais à celle du Contrôleur, c'est pourquoi ce bout de code doit être déplacé dans une action dédiée.

Le problème qui se pose ici est que le formulaire de changement de langue appartient au layout car il doit être présent sur chaque page du site. La solution qui en découle est alors d'instancier la classe `sfFormLanguage` dans chacune des actions du site, ce qui est loin d'être très pratique et *DRY*… Heureusement avec Symfony, chaque problème a sa solution propre, et dans le cas présent il s'agit de faire appel aux composants.

Que sont les composants dans Symfony ?

Dans Symfony, un composant est en réalité un template partiel auquel est attaché du code métier, ce qui est finalement comparable à une action allégée. Comme les actions, les composants dépendent tous d'un module, ce qui implique qu'il faille créer un fichier `components.class.php` dans le répertoire `actions/` d'un module.

Ce fichier contiendra la classe `nomDuModuleComponents` des différents composants rattachés à ce module. Enfin, chaque composant développé dans cette classe est en réalité une méthode avec du code métier à laquelle est associé un template partiel du même nom que le composant.

L'étape finale de mise en place d'un composant est bien évidemment son appel depuis un template qui se réalise à l'aide du helper `include_component()` comme le montre l'exemple de code ci-dessous.

Intégration du composant language dans le fichier apps/frontend/templates/layout.php

```
<div id="footer">
  <div class="content">
    <!-- footer content -->

    <?php include_component('language', 'language') ?>
  </div>
</div>
```

Le helper `include_component()` accepte deux arguments obligatoires qui correspondent respectivement au nom du module dans lequel se situe le composant et enfin l'action à appeler dans la classe de composants. Dans le

cas présent, le composant ira chercher l'action `language` des composants du module `language` qui n'existe pas encore. Ce helper est capable d'accueillir un troisième paramètre facultatif qui doit être un tableau associatif de couples variable/valeur nécessaires au bon fonctionnement du composant.

Implémenter le composant de changement de langue

La section suivante est relativement théorique, c'est pour cette raison qu'il convient de la mettre en pratique en développant pas à pas le composant du formulaire de changement de langue. Pour y parvenir, cinq étapes successives sont à franchir :

1. générer le module `language` et créer le fichier `components.class.php` ;
2. développer la classe `languageComponent` et sa méthode `executeLanguage()` ;
3. créer le template du composant ;
4. déclarer une nouvelle route dédiée pour l'action du formulaire ;
5. développer la logique métier du changement de la culture de l'utilisateur.

La première étape consiste donc à générer le squelette du module `language` à l'aide de la tâche Symfony `generate:module`.

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

Une fois le module complètement généré, le fichier des composants `components.class.php` doit être créé manuellement dans le répertoire `actions/` du module `language`, dans la mesure où la tâche `generate:module` ne le crée pas. Ce fichier contient la classe `languageComponents` dont le code figure ci-dessous.

Contenu du fichier 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 on peut le constater ici, une classe de composants se déclare globalement de la même façon qu'une classe d'actions. Le nom de la classe est composé du nom du module suffixé par `Components` et la classe dérivée est cette fois-ci `sfComponents`. Les actions, quant à elles, s'écri-

vent toujours de la même manière. Ici, le composant `language` ne fait ni plus ni moins qu'instancier la classe `sfFormLanguage` avec ces paramètres, puis transmet l'objet `$form` à son template associé.

La réalisation du template du composant constitue la troisième étape du processus de création d'un composant. Comme pour les templates des actions, les templates des composants sont soumis à une convention de nommage particulière. Le template d'un composant est en fait un template partiel portant le nom de l'action appelée. Le code ci-dessous est le contenu du fichier `_language.php` à créer dans le dossier `templates/` du module `language`.

Contenu du template partiel du composant language dans le fichier apps/frontend/modules/language/templates/_language.php

```
<form action="<?php echo url_for('@change_language') ?>">
  <?php echo $form ?><input type="submit" value="ok" />
</form>
```

Le composant est maintenant entièrement prêt. L'étape suivante consiste à le rendre fonctionnel à l'aide de l'action exécutée à l'appel de la route `change_language`. Dans un premier temps, celle-ci doit être déclarée dans le fichier `routing.yml` de l'application frontend.

Route change_language à ajouter au fichier apps/frontend/config/routing.yml

```
change_language:
  url:   /change_language
  param: { module: language, action: changeLanguage }
```

La méthode `changeLanguage` du module `language` a pour rôle de traiter la valeur transmise par le formulaire de changement de langue et de fixer la valeur de la culture de l'utilisateur, avant de rediriger ce dernier vers la page d'accueil traduite dans la langue qu'il a choisie.

Implémentation de l'action changeLanguage dans le fichier 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');
  }
}
```

Ici, c'est en réalité la méthode `process()` de l'objet `sfFormLanguage` qui se charge de contrôler la valeur soumise par l'utilisateur dans la requête, et d'affecter la nouvelle culture à l'objet `myUser` qui lui est passé en guise de premier paramètre du constructeur. La capture d'écran ci-dessous donne le résultat final de l'intégration du composant `language` dans le pied de page de Jobeet.

Figure 18–1 Rendu final du formulaire de changement de langue dans le pied de page de Jobeet

Découvrir les outils d'internationalisation de Symfony

Le framework Symfony recèle d'outils divers et variés pour simplifier la création d'applications web internationalisées et régionalisées. Ce lot d'outils présentés dans les sections qui suivent comprend notamment le support des encodages, la traduction des contenus dans les templates, l'extraction de contenus internationalisés des templates, des helpers de formatage de dates, heures, nombres et monnaies, ou encore le support intégral des tables de traduction pour les objets Doctrine en base de données.

Paramétrer le support des langues, jeux de caractères et encodages

La monde comprend plus d'une centaine de langues officielles toutes autant variées les unes que les autres étant donné leurs spécificités typographiques (sens de lecture, lettres, symboles, ponctuation…). Cette large variété de langues explique pourquoi, dans le monde informatique, il existe autant de jeux de caractères différents entre les langues.

La langue anglaise est sans aucun doute la plus simple de toutes puisqu'elle fait appel uniquement à des caractères de la table ASCII (se prononce « ASKI ») tandis que le français est, quant à lui, un peu plus complexe en raison de ses caractères accentués tels que *é*, *ù*, *î*… D'autres langues comme le russe, le chinois, ou l'arabe sont encore plus complexes dans la mesure où leurs caractères et/ou symboles typographiques sortent littéralement du cadre de la table ASCII, ce qui implique que ces langues sont définies avec des jeux de caractères différents.

Lorsqu'il s'agit de manipuler des données internationalisées, il est une bonne pratique d'utiliser la norme Unicode. Le concept sous-jacent d'Unicode est d'établir un jeu de caractères universel capable de contenir tous les caractères de toutes les langues. Cependant, cela a un inconvénient notable car un seul caractère codé avec Unicode peut être représenté avec plus de 21 bits. Par conséquent, pour le web, c'est l'UTF-8 qui est retenu car il fait correspondre les numéros de chaque caractère Unicode avec des séquences d'octets de longueur variable (de 1 à 4 octets). En UTF-8, la plupart des langues ont leurs caractères codés avec moins de 3 bits.

Le framework Symfony s'appuie par défaut sur l'encodage UTF-8, défini dans le fichier de configuration `settings.yml` de chaque application.

Configuration de l'encodage par défaut de Symfony dans le fichier apps/frontend/config/settings.yml

```
all:
  .settings:
    charset: utf-8
```

De plus, par défaut, la couche d'internationalisation de Symfony est désactivée afin d'éviter de charger des composants si l'application n'a pas vocation à être traduite en plusieurs langues. Pour l'activer, il suffit d'agir sur la directive de configuration `i18n` du fichier de configuration `settings.yml` de l'application en plaçant sa valeur à `on`.

Activation de l'internationalisation dans le fichier de configuration apps/frontend/config/settings.yml

```
all:
  .settings:
    i18n: on
```

Maintenant que l'application est convenablement paramétrée pour accueillir du contenu internationalisé, il convient d'étudier les outils offerts par Symfony qui permettent de simplifier la traduction de données statiques, à commencer par les templates.

Traduire les contenus statiques des templates

Les pages web contiennent de nombreux contenus statiques (des mots ou des phrases) et dynamiques (des motifs de phrases) qu'il convient de traduire. Ces informations sont généralement présentes en dur dans les templates mais elles peuvent également provenir des classes de modèles ou des actions. C'est le cas par exemple des intitulés et des messages d'erreurs d'un formulaire, ou encore des messages éphémères qui sont affichés en guise de feedback à l'utilisateur après qu'il a exécuté une action critique.

Le framework Symfony intègre une série de helpers qui facilitent la traduction de tous ces types de contenus destinés aux templates. Le premier et sans doute le principal d'entre eux est la fonction `__()`.

Utiliser le helper de traduction __()

Dans un template, toutes les chaînes de caractères qui dépendent de la langue doivent être encapsulées avec le helper `__()`, qui s'écrit seulement avec deux tirets soulignés (underscore). Cette fonction fait partie intégrante des helpers du groupe I18N, qui contient un jeu de helpers destinés à faciliter la gestion des contenus internationalisés dans les templates.

Charger automatiquement le helper __()

Le groupe de helpers I18N n'est pas chargé par défaut par le framework pour des raisons de performance. Le groupe I18N doit donc être chargé manuellement, soit localement dans chaque template qui en fait usage via l'utilisation du helper `use_helper('I18N')` qui a été présenté à l'occasion du chargement des helpers du groupe `Text`, soit globalement pour toute l'application en l'ajoutant à la liste des helpers standards dans la directive de configuration `standard_helpers` du fichier de configuration `settings.yml`.

Ajout du groupe de helpers I18N dans le fichier apps/frontend/config/settings.yml

```
all:
  .settings:
    standard_helpers: [Partial, Cache, I18N]
```

Après une régénération du cache de Symfony, toutes les pages de Jobeet pourront faire appel au helper `__()`. La section ci-après montre dans le détail comment s'utilise ce helper.

Traduire les contenus statiques dans les templates

Le fonctionnement de cette fonction est extrêmement simple puisqu'il s'agit d'afficher tous les contenus statiques à l'aide de ce helper comme le

CHOIX DE CONCEPTION Utiliser une chaîne ou un identifiant unique en guise de paramètre

Le helper `__()` accepte en guise de paramètre aussi bien une phrase de la langue source par défaut qu'un identifiant unique pour chaque traduction. Ceci est en fait une simple question de goût pour le développeur. Dans le cadre de Jobeet, c'est la première méthode qui est utilisée car elle permet de rendre les templates bien plus lisibles que la seconde.

CHOIX DE CONCEPTION
Utiliser d'autres types de catalogue

D'autres catalogues de stockage des traductions existent dans Symfony. Il s'agit par exemple de gettext, MySQL ou SQLite. Comme toujours, il suffit de regarder dans la documentation ou dans l'API I18N pour davantage de détails.

TECHNOLOGIE Le format XLIFF

XLIFF est l'acronyme de *XML Localization Interchange File Format*. Il s'agit d'un format standard de données reposant sur le format XML qui permet de faciliter le stockage des données visant à être internationalisées. Le format XLIFF a été standardisé en 2002 par l'OASIS, l'organisation qui se charge de standardiser les formats de structuration de l'information pour les domaines de l'informatique, du e-business, de la sécurité, du gouvernement, des services web… La spécification actuelle de XLIFF établit les attributs et les éléments nécessaires d'aide à la localisation.

présente le code ci-dessous. Il s'agit ici du pied de page du template du layout de l'application de Jobeet dans lequel tous les contenus statiques à traduire ont été encapsulés dans un appel à `__()`, au lieu d'être inscrits directement en dur sous forme de données HTML brutes.

Utilisation du helper __() dans le fichier 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?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>
```

Lorsque le framework évalue un template, et à chaque fois que le helper `__()` est appelé, Symfony recherche une traduction pour la culture de l'utilisateur courant. Si la traduction est trouvée dans le catalogue, elle est renvoyée, sinon c'est la chaîne passée en premier argument qui est renvoyée à la place.

Toutes les traductions sont stockées dans un catalogue. Le framework d'internationalisation de Symfony fournit différentes stratégies pour sauvegarder les traductions. Pour Jobeet, c'est le stockage dans des fichiers XLIFF (*XML Localization Interchange File Format*) qui est choisi dans la mesure où c'est le moyen le plus flexible et le plus couramment employé dans les projets Symfony. C'est d'ailleurs le format de stockage des traductions du générateur de backoffice et de la plupart des plug-ins.

Extraire les contenus internationalisés vers un catalogue XLIFF

L'écriture manuelle de catalogues XLIFF est relativement fastidieuse, c'est pourquoi le framework Symfony fournit la tâche automatique `i18n:extract` pour en simplifier la génération en analysant un à un les templates ayant des contenus internationalisables.

```
$ php symfony i18n:extract frontend fr --auto-save
```

La tâche `i18n:extract` trouve toutes les chaînes de caractères qui ont besoin d'être traduites en langue française (`fr`) dans l'application frontend, puis crée ou met à jour le catalogue correspondant. L'option `--auto-save` force la tâche à sauvegarder dans le catalogue les nouvelles chaînes internationnalisables qu'elle trouve. Il est également possible d'utiliser l'option `--auto-delete` qui supprime du catalogue toutes les chaînes qui n'existent plus dans les templates.

Pour Jobeet, le résultat d'exécution de cette tâche automatique produit le fichier XML XLIFF suivant :

Contenu du fichier XLIFF 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>
```

TECHNOLOGIE
Outils d'analyse de fichiers XLIFF

Comme XLIFF est un format standard, il existe un nombre important d'outils capables de simplifier le processus de traduction d'une application. C'est le cas par exemple du projet libre Java « Open Language Tools » qui intègre un éditeur de code XLIFF.

ASTUCE **Surcharger les traductions à plusieurs niveaux**

XLIFF est un format basé sur des fichiers. De ce fait, les mêmes règles de priorité et de fusion que celles des autres fichiers de configuration de Symfony peuvent lui être applicables. Les fichiers I18N peuvent ainsi exister au niveau du projet, d'une application ou bien d'un module, et ce sont les plus spécifiques qui redéfinissent les traductions présentes aux niveaux globaux. Le niveau du module est donc prioritaire sur celui de l'application, qui lui même est prioritaire sur celui du projet.

Chaque traduction est gérée par une balise `<trans-unit>` qui dispose obligatoirement d'un attribut `id` en guise d'identifiant unique. Il suffit alors d'ajouter manuellement toutes les traductions pour la langue française à l'intérieur de chaque balise `<target>` correspondante. Ainsi, cela conduit à un fichier tel que celui ci-dessous.

Contenu du fichier XLIFF 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>
```

La section suivante aborde un nouveau point intéressant du processus de traduction. Il s'agit des contenus dynamiques. En effet, seuls les contenus statiques ont été traduits pour l'instant mais il est aussi fréquent d'avoir à traduire des contenus qui intègrent des valeurs dynamiques.

Traduire des contenus dynamiques

Le principe global sous-jacent de l'internationalisation est de traduire des phrases entières, comme cela a été expliqué plus haut. Cependant, il est fréquent d'avoir à traduire des phrases qui embarquent des valeurs dynamiques et qui reposent sur un motif particulier. Par exemple,

lorsqu'il s'agit d'afficher un nombre au milieu d'une chaîne internationalisable telle que « il y a 10 personnes connectées au site ».

Le cas des chaînes dynamiques simples

Dans Jobeet, c'est aussi le cas avec la page d'accueil qui inclut un lien « more… » dont l'affichage final ressemble au motif suivant : « and 12 more… ». Le code ci-dessous est l'implémentation actuelle de ce lien dans Jobeet.

Extrait du contenu du fichier apps/frontend/modules/job/templates/indexSuccess.php

```
<div class="more_jobs">
  and <?php echo link_to($count, 'category', $category) ?>
more...
</div>
```

Le nombre d'offres d'emploi est une variable qui devrait être remplacée par un emplacement (*placeholder*) pour en simplifier sa traduction. Le framework Symfony supporte ce type de phrases composées de valeurs dynamiques comme le montre le code ci-dessous.

Extrait du contenu du fichier 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 à présent « `and %count% more...` », et l'emplacement `%count%` sera remplacé automatiquement par le nombre réel d'offres supplémentaires à l'instant t, grâce au second paramètre facultatif du helper `__()`. Il ne reste finalement qu'à ajouter cette nouvelle chaîne à traduire au catalogue de traductions françaises, soit en l'ajoutant manuellement dans le fichier `messages.xml`, soit en réexécutant la tâche `i18n:extract` pour le mettre à jour automatiquement.

```
$ php symfony i18n:extract frontend fr --auto-save
```

Après exécution de cette commande, le fichier XLIFF accueille une nouvelle traduction pour cette chaîne dont le motif est le suivant :

```
<trans-unit id="5">
  <source>and %count% more...</source>
  <target>et %count% autres...</target>
</trans-unit>
```

La seule obligation dans la chaîne traduite est de réutiliser l'emplacement `%count%`.

Traduire des contenus pluriels à partir du helper format_number_choice()

D'autres contenus internationalisables sont un peu plus complexes dans la mesure où ils impliquent des pluriels. D'après la valeur de certains nombres, la phrase change, mais pas nécessairement de la même manière pour toutes les langues. Certaines langues comme le russe ou le polonais ont des règles de grammaire très complexes pour gérer les pluriels. Sur la page de détail d'une catégorie, le nombre d'offres dans la catégorie courante est affiché de la manière suivante.

```
<strong><?php echo $pager->getNbResults() ?></strong> jobs in
this category
```

Lorsqu'une phrase possède différentes traductions en fonction d'un nombre, le helper `format_number_choice()` prend le relais pour prendre en charge la bonne traduction à afficher.

```
<?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>'.$pager->getNbResults().'</
strong>'),
    $pager->getNbResults()
  )
?>
```

Le helper `format_number_choice` accepte trois arguments :

- la chaîne à afficher qui dépend du nombre ;
- un tableau des valeurs des emplacements à remplacer ;
- le nombre à tester pour déterminer quelle traduction afficher.

La chaîne qui décrit les différentes traductions en fonction du nombre est formatée de la manière suivante :

- chaque possibilité est séparée par un caractère pipe (|) ;
- chaque chaîne est composée d'un intervalle de valeurs numéraires suivi par la traduction elle-même.

L'intervalle peut décrire n'importe quel type de suite ou d'intervalle de nombres :

- `[1,2]` : accepte toutes les valeurs comprises entre 1 et 2, bornes incluses ;
- `(1,2)` : accepte toutes les valeurs comprises entre 1 et 2, bornes exclues ;
- `{1,2,3,4}` : accepte uniquement les valeurs définies dans cet ensemble ;

- `[-Inf,0)` : accepte les valeurs supérieures ou égales à l'infini négatif et strictement inférieures à 0 ;
- `{n: n % 10 > 1 && n % 10 < 5}` : correspond aux nombres comme 2, 3, 4, 22, 23, 24.

Traduire ce type de chaîne est similaire aux autres messages de traduction :

```
<trans-unit id="6">
  <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 tous les moyens de traduction des chaînes statiques et dynamiques ont été présentés, il ne reste plus qu'à prendre le temps pour appréhender le helper `__()` en traduisant tous les messages de l'application frontend. Pour la suite de ce chapitre, l'application backend ne sera pas internationalisée.

Traduire les contenus propres aux formulaires

Les classes de formulaire contiennent plusieurs chaînes qui ont besoin d'être traduites comme les intitulés des champs, les messages d'erreurs et les messages d'aide. Heureusement, toutes ces chaînes sont automatiquement internationalisées par Symfony. Par conséquent, il suffit uniquement de spécifier les traductions dans le fichier XLIFF.

À RETENIR Limites de la tâche automatique i18n:extract

Malheureusement, la commande `i18n:extract` a ses limites puisqu'elle n'analyse pas encore les classes de formulaire à la recherche de chaînes non traduites.

Activer la traduction des objets Doctrine

L'un des sujets les plus délicats à traiter lorsqu'on manipule des données internationalisables dans une application dynamique concerne bien évidemment les enregistrements de la base de données. Dans cette section, il s'agit de découvrir de quelle manière le framework Symfony et l'ORM Doctrine simplifient la gestion des contenus internationalisés en base de données.

Pour l'application Jobeet, il ne sera pas utile d'internationaliser toutes les tables dans la mesure où cela n'a pas de véritable sens de demander aux auteurs d'offres de traduire leurs propres annonces. Néanmoins, la traduction de la table des catégories semble légitime.

Internationaliser le modèle JobeetCategory de la base

Le plug-in Doctrine supporte nativement les tables d'internationalisation. Pour chaque table qui contient des données régionalisées, deux tables sont finalement nécessaires. En effet, la première stocke les valeurs indépendantes de l'I18N pour l'enregistrement donné, tandis que la seconde sert à conserver les valeurs des champs internationalisés de ce même enregistrement. Les deux tables sont reliées entre elles par une relation dite « one-to-many ».

L'internationalisation de la table des catégories nécessite de mettre à jour le modèle `JobeetCategory` comme le montre le schéma ci-dessous.

Ajout du comportement I18N au modèle JobeetCategory dans le fichier 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 activant le comportement `I18n`, un nouveau modèle intitulé `JobeetCategoryTranslation` sera automatiquement créé et les champs localisables seront déplacés vers ce modèle. De plus, il faut remarquer que le comportement `Sluggable` a été déporté vers le modèle d'internationalisation `JobeetCategoryTranslation`. L'option `uniqueBy` indique au comportement `Sluggable` quels champs déterminent si un slug est unique ou non. Dans le cas présent, il s'agit de rendre unique chaque paire langue/slug.

Mettre à jour les données initiales de test

Avant de reconstruire tout le modèle, il convient de mettre à jour les données initiales de l'application puisque le champ `name` d'une catégorie n'appartient plus directement au modèle `JobeetCategory` mais à l'objet `JobeetCategoryTranslation`. Le code ci-dessous donne le contenu du fichier de données initiales des catégories qui seront rechargées en base de données à la reconstruction de tout le modèle.

Contenu du fichier 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
```

Surcharger la méthode findOneBySlug() du modèle JobeetCategoryTable

La méthode `findOneBySlug()` de la classe `JobeetCategoryTable` doit être redéfinie. En effet, depuis que Doctrine fournit quelques *finders* magiques pour chaque colonne d'un modèle, il suffit de créer une méthode `findOneBySlug()` qui surcharge le comportement initial du finder Doctrine. Pour ce faire, cette méthode doit arborer quelques changements supplémentaires afin que la catégorie soit retrouvée à partir du slug anglais présent dans la table `JobeetCategoryTranslation`.

Implémentation de la méthode findOneBySlug() dans le fichier 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();
}
```

À RETENIR La tâche doctrine:build-all-reload

Comme la commande `doctrine:build-all-reload` supprime toutes les tables et données de la base de données, il ne faut pas oublier de recréer un utilisateur pour accéder à l'espace d'administration de Jobeet à l'aide de la tâche `guard:create-user` présentée dans les chapitres précédents. Alternativement, il paraît aussi astucieux d'ajouter un fichier de données initiales contenant les informations de l'utilisateur qui seront automatiquement chargées en base de données.

La prochaine étape consiste à présent à reconstruire tout le modèle à l'aide de la tâche automatique `doctrine:build-all`.

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

Méthodes raccourcies du comportement I18N

Lorsque le comportement I18N est attaché à une classe de modèle comme celle des catégories par exemple, des méthodes raccourcies (dites « proxies ») entre l'objet `JobeetCategory` et les objets `JobeetCategoryTranslation` associés sont créées. Grâce à elles, les anciennes méthodes pour retrouver le nom de la catégorie continuent de fonctionner en se fondant sur la valeur de la culture courante.

```
$category = new JobeetCategory();
$category->setName('foo'); // définit le nom pour la culture
courante
$category->getName(); // retourne le nom pour la culture
courante

$this->getUser()->setCulture('fr'); // depuis une classe
d'actions

$category->setName('foo'); // définit le nom en français
echo $category->getName(); // retourne le nom en français
```

Pour réduire le nombre de requêtes à la base de données, il convient de joindre la table `JobeetCategoryTranslation` dans les requêtes SQL. Cela permettra de récupérer l'objet principal et ses informations internationalisées en 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 ajoutera automatiquement la condition à la clause `ON` de la requête SQL, ce qui se traduit au final par la requête SQL suivante :

```
LEFT JOIN c.Translation t ON c.id = t.id AND t.lang = ?
```

Mettre à jour le modèle et la route de la catégorie

Puisque d'une part la route qui mène à la catégorie est liée au modèle `JobeetCategory`, et que d'autre part le slug est maintenant un champ de la table `JobeetCategoryTranslation`, la route n'est plus capable de

retrouver l'objet `category` automatiquement. Pour aider le système de routage de Symfony, une nouvelle méthode doit être ajoutée au modèle afin de se charger de la récupération de l'objet.

Implémenter la méthode findOneBySlugAndCulture() du modèle JobeetCategoryTable

La méthode `findOneBySlug()` a déjà été redéfinie plus haut, mais a vocation à être factorisée davantage afin que les nouvelles méthodes puissent être partagées. L'objectif est de créer deux nouvelles méthodes `findOneBySlugAndCulture()` et `doSelectForSlug()`, puis de changer `findOneBySlug()` pour utiliser simplement la méthode `findOneBySlugAndCulture()`.

Implémentation des nouvelles méthodes dans la classe JobeetCategoryTable du fichier 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');
}

// ...
```

Mise à jour de la route category de l'application frontend

Les méthodes de la classe `JobeetCategoryTable` sont maintenant idéalement factorisées, ce qui permet désormais à la route de retrouver l'objet `JobeetCategory` auquel elle est liée. Pour ce faire, l'option `method` de la route Doctrine doit être éditée pour lui indiquer qu'elle aura recours à la méthode `getForSlug()` pour récupérer l'objet associé.

Route category du fichier 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)
```

Pour finir, les données initiales doivent être renouvelées dans la base de données afin de régénérer les champs internationalisés.

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

La route `category` est désormais entièrement internationalisée et embarque les slugs appropriés en fonction de la langue du site.

```
/frontend_dev.php/fr/category/programmation
/frontend_dev.php/en/category/programming
```

Champs internationalisés dans un formulaire Doctrine

Internationaliser le formulaire d'édition d'une catégorie dans le backoffice

Suite à tous les ajustements qui ont été opérés jusqu'à présent, les catégories sont désormais entièrement internationalisées mais ne bénéficient pas encore d'un moyen pour gérer l'ensemble des champs traduits. L'interface d'administration actuelle permet uniquement d'éditer les champs d'une catégorie correspondante à la culture de l'utilisateur. Or, il paraît pertinent de permettre à l'administrateur du site d'agir sur l'ensemble des champs internationalisés du formulaire en une seule passe comme cela figure sur la capture d'écran ci-contre.

Utiliser la méthode embedI18n() de l'objet sfFormDoctrine

Tous les formulaires Doctrine supportent nativement les relations avec les tables d'internationalisation des objets qu'ils permettent de manipuler. En effet, l'objet `sfFormDoctrine` dispose de la méthode `embedI18n()` qui ajoute automatiquement le contrôle de tous les champs internationalisables d'un objet. Son usage est extrêmement simple puisqu'il s'agit de l'appeler dans la méthode `configure()` du formulaire en lui passant en paramètre un tableau des cultures à intégrer au formulaire. Le code ci-dessous correspond à la classe de formulaire `JobeetCategoryForm` qui fait appel à cette méthode pour afficher les champs traduits pour les langues française et anglaise.

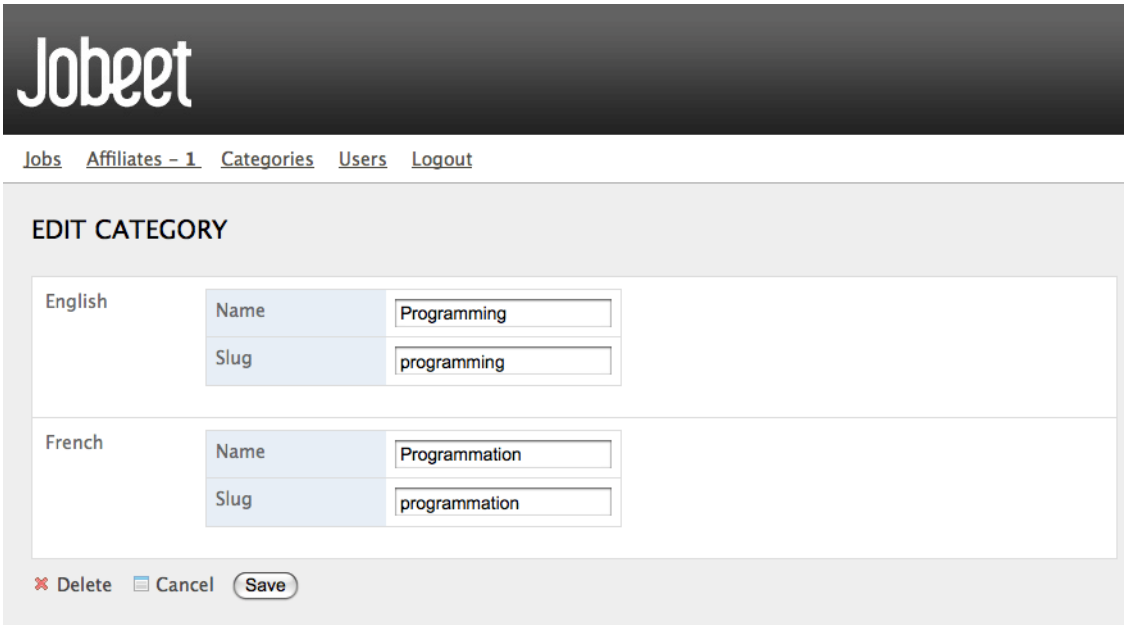


Figure 18–2 Formulaire d'édition des champs internationalisés d'une catégorie

Implémentation de la méthode embedI18n() dans la classe JobeetCategoryForm du fichier 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');
  }
}
```

Internationalisation de l'interface du générateur d'administration

L'interface issue du générateur d'administration supporte nativement l'internationalisation. Le framework Symfony est livré avec les fichiers de traduction XLIFF de l'interface d'administration dans plus de vingt langues différentes grâce à l'effort de la communauté. Il est donc désormais très facile d'ajouter de nouvelles traductions ou bien de personnaliser les intitulés existants.

Pour y parvenir, il suffit de copier le fichier des traductions de la langue à personnaliser dans le répertoire `i18n/` de l'application. Les fichiers de traduction de Symfony pour le plug-in `sfDoctrinePlugin` se situent dans le répertoire `lib/vendor/symfony/lib/plugins/sfDoctrinePlugin/i18n/` du projet Jobeet. Comme le fichier de l'application est fusionné avec celui de Symfony, il suffit de garder uniquement les traductions modifiées dans le nouveau fichier de traduction.

Forcer l'utilisation d'un autre catalogue de traductions

Il est intéressant de remarquer que les fichiers de traduction du générateur d'interface d'administration sont nommés suivant le format `sf_admin.fr.xml`, au lieu de `fr/messages.xml`. En fait, `messages` est le nom du catalogue utilisé par défaut dans Symfony mais il peut bien sûr être modifié afin de permettre une meilleure séparation des différentes parties de l'application. Utiliser un catalogue spécifique plutôt que celui par défaut nécessite de l'indiquer explicitement au helper `__()` à l'aide de son troisième argument facultatif.

```
<?php echo __('About Jobeet', array(), 'jobeet') ?>
```

Dans l'appel au helper `__()` ci-dessus, Symfony cherchera la chaîne « About Jobeet » dans le catalogue `jobeet`.

Tester l'application pour valider le processus de migration de l'I18N

Corriger les tests fonctionnels fait partie intégrante du processus de migration vers une interface internationalisée. Dans un premier temps, les fichiers de données de tests pour les catégories doivent être mis à jour en récupérant celles qui se trouvent dans le fichier `test/fixtures/categories.yml`. Puis l'intégralité du modèle et de la base de données de test doit à son tour être régénérée pour prendre en compte toutes les modifications réalisées jusqu'à présent.

```
$ php symfony doctrine:build-all-reload --no-confirmation
  ➥ --env=test
```

Enfin, l'exécution de toute la suite de tests unitaires et fonctionnels indiquera si l'application se comporte toujours aussi bien ou non. Si des tests échouent, c'est qu'une régression a probablement été engendrée quelque part lors du processus de migration.

```
$ php symfony test:all
```

À RETENIR Supprimer les squelettes de fichiers de tests autogénérés

Lorsque l'interface d'administration de Jobeet a été développée, aucun test fonctionnel n'a été écrit. Néanmoins, à chaque fois qu'un nouveau module est généré à l'aide de la ligne de commande de Symfony, le framework en profite pour créer des squelettes de fichiers de tests. Ces derniers peuvent être supprimés du projet en toute sécurité.

Découvrir les outils de localisation de Symfony

La fin de ce chapitre arrive pratiquement à son terme et il reste pourtant un point important qui n'a pas encore été traité dans le processus d'internationalisation d'une application web : la localisation. En effet, la localisation est la partie de l'internationalisation relative à toutes les questions de formatage de données propres à la région de l'utilisateur comme les nombres, les dates, les heures ou bien encore les devises monétaires.

Régionaliser les formats de données dans les templates

Supporter différentes langues signifie aussi supporter différentes manières de formater les dates et les nombres. Pour les templates, le framework Symfony met à disposition un jeu de helpers qui permettent d'aider à prendre en compte toutes ces différences relatives à la culture courante de l'utilisateur.

Les helpers du groupe Date

Le groupe de helpers `Date` fournit cinq fonctions pour assurer le formatage des dates et des heures dans les templates.

Tableau 18–1 Liste des fonctions de formatage de dates et heures du groupe de helpers Date

Nom du helper	Description
`format_date()`	Formate une date
`format_datetime()`	Formate une date et le temps (heures, minutes, secondes…)
`time_ago_in_words()`	Affiche le temps passé depuis une date jusqu'à maintenant en toutes lettres
`distance_of_time_in_words()`	Affiche le temps passé entre deux dates en toutes lettres
`format_daterange()`	Formate un intervalle de dates

Le détail de ces helpers est disponible dans l'API de Symfony à l'adresse : http://www.symfony-project.org/api/1_2/DateHelper

Les helpers du groupe Number

Le groupe de helpers `Number` fournit deux fonctions pour assurer le formatage des nombres et devises dans les templates.

Tableau 18–2 Liste des fonctions de formatage de nombres et devises du groupe de helpers Number

Nom du helper	Description
`format_number()`	Formate un nombre
`format_currency()`	Formate une devise monétaire

Le détail de ces helpers est disponible dans l'API de Symfony à l'adresse : http://www.symfony-project.org/api/1_2/NumberHelper

Les helpers du groupe I18N

Le groupe de helpers `I18N` fournit deux fonctions pour assurer le formatage des noms de pays et langues dans les templates.

Tableau 18–3 Liste des fonctions de formatage des noms de pays et langues du groupe de helpers I18N

Nom du helper	Description
`format_country()`	Affiche le nom d'un pays
`format_language()`	Affiche le nom d'une langue

Le détail de ces helpers est disponible dans l'API de Symfony à l'adresse : http://www.symfony-project.org/api/1_2/I18NHelper

Régionaliser les formats de données dans les formulaires

Le framework de formulaires de Symfony fournit également plusieurs widgets et validateurs pour gérer les différentes données localisées. Le tableau ci-après donne le nom ainsi qu'une description de chacun d'eux.

Tableau 18–4 Widgets et validateurs de données localisées dans les formulaires

Nom du composant	Description
`sfWidgetFormI18nDate`	Génère un widget de saisie de date
`sfWidgetFormI18nDateTime`	Génère un widget de saisie de date et de temps
`sfWidgetFormI18nTime`	Génère un widget de saisie de temps
`sfWidgetFormI18nSelectCountry`	Génère une liste déroulante de pays
`sfWidgetFormI18nSelectCurrency`	Génère une liste déroulante de devises monétaires
`sfWidgetFormI18nSelectLanguage`	Génère une liste déroulante de langues
`sfValidatorI18nChoiceCountry`	Valide la valeur d'un pays
`sfValidatorI18nChoiceLanguage`	Valide la valeur d'une langue

En résumé…

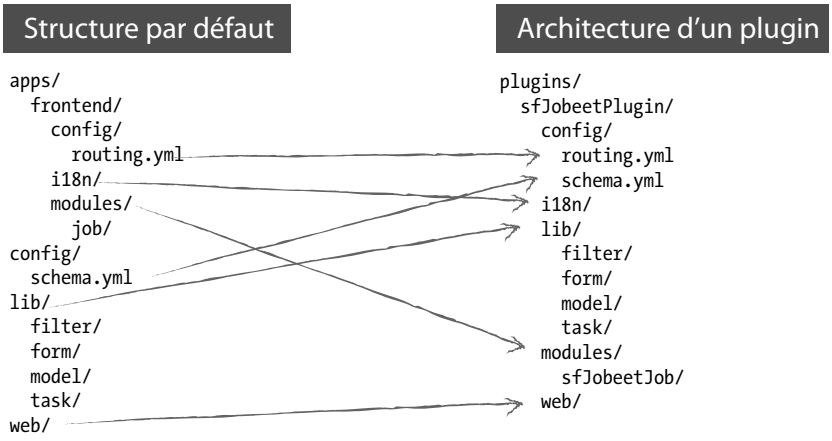
L'internationalisation et la localisation sont des concepts parfaitement intégrés dans Symfony. Le processus d'internationalisation d'un site Internet à destination des utilisateurs est extrêmement aisé dans la mesure où Symfony fournit tous les outils nécessaires ainsi qu'une interface en ligne de commande pour en accélérer le développement.

Ce chapitre a permis de découvrir l'ensemble des outils qu'offre Symfony pour simplifier l'internationalisation d'une application. Ce large panel d'outils comprend à la fois des catalogues de traduction XLIFF et de nombreux jeux de *helpers* pour formater des données (nombres, dates, heures, langues, devises monétaires…), traduire des contenus textuels statiques et dynamiques ou encore de gérer les pluriels.

Avec l'intégration de l'ORM Doctrine, l'internationalisation des objets en base de données est grandement facilitée puisque Symfony et Doctrine fournissent les APIs adéquates pour prendre automatiquement en charge la manipulation des données à traduire, notamment lorsqu'il s'agit de les manipuler par le biais des formulaires.

Le chapitre suivant est un peu spécial puisqu'il y est question de déplacer la plupart des fichiers de Jobeet à l'intérieur de plug-ins personnalisés. Les plug-ins constituent en effet une approche différente pour organiser un projet Symfony. Ces prochaines pages seront donc l'occasion de découvrir l'ensemble des multiples avantages qu'offre une telle approche organisationnelle…

chapitre 19


Structure par défaut
apps/
frontend/
config/
routing.yml
i18n/
modules/
job/
config/
schema.yml
lib/
filter/
form/
model/
task/
web/
Architecture d'un plugin
plugins/
sfJobeetPlugin/
config/
routing.yml
schema.yml
i18n/
lib/
filter/
form/
model/
task/
modules/
sfJobeetJob/
web/

Les plug-ins

MOTS-CLÉS :

- Plug-ins
- Internationalisation
- Routage

Le framework Symfony bénéficie d'une communauté qui contribue activement au développement du projet à travers la création et la publication de plug-ins. Les plug-ins sont des unités fonctionnelles indépendantes du projet qui remplissent des besoins spécifiques (authentification, traitement d'images, API de manipulation d'un service web…) afin d'empêcher le développeur de réinventer une nouvelle fois la roue et d'accélérer la mise en place de son projet.

Le chapitre précédent a abordé dans son intégralité le vaste sujet de l'internationalisation et de la localisation d'une application web. Ces deux concepts sont nativement supportés à tous les niveaux (base de données, routage, catalogue de traductions…) grâce aux nombreux outils qu'offre le framework Symfony. Aborder ce sujet fut également l'occasion d'installer le plug-in `sfFormExtraPlugin` et d'en découvrir certaines fonctionnalités. Ce dix-neuvième chapitre s'intéresse tout particulièrement aux plug-ins : ce qu'ils sont, à quoi ils servent, comment les développer et les diffuser sur le site Internet de Symfony...

Qu'est-ce qu'un plug-in dans Symfony ?

Les plug-ins Symfony

Un plug-in Symfony offre une manière différente de centraliser et de distribuer un sous-ensemble des fichiers d'un projet. Au même titre qu'un projet, un plug-in est capable d'embarquer des classes, des helpers, des fichiers de configuration, des tâches automatisées, des modules fonctionnels, des schémas de description d'un modèle de données, ou encore des ressources pour le web (feuilles de style en cascade, JavaScripts, animations Flash…).

Une section prochaine de ce chapitre révèlera qu'en fait, un plug-in est structuré quasiment de la même manière qu'un projet, voire une application. En somme, un plug-in peut aussi bien transporter uniquement un jeu restreint de fichiers (des classes par exemple) qu'embarquer une application fonctionnelle complète comme un forum de discussion ou un CMS.

Les plug-ins privés

Le premier usage des plug-ins est avant tout de faciliter le partage de code entre les applications, et dans le meilleur des cas, entre les projets. Il est important de se rappeler que les applications Symfony d'un même projet ne partagent que le modèle et quelques classes et fichiers de configuration. Les plug-ins, quant à eux, fournissent un moyen de partager davantage de composants entre les applications.

Lorsque naît le besoin de réutiliser le même schéma de données pour différents projets, ou pour les mêmes modules, cela signifie qu'il est préférable de le déplacer vers un plug-in. En pratique, un plug-in n'est finalement qu'un simple répertoire ; c'est pour cette raison qu'il devient alors possible de l'externaliser aussi facilement, par exemple en créant un dépôt SVN dédié ou simplement en copiant les fichiers d'un projet dans un autre.

ASTUCE

Installer ses propres plug-ins privés

La tâche automatique `plugin:install` est capable d'installer des plug-ins privés à condition que ces derniers aient été réalisés comme il se doit, et qu'un canal privé de plug-ins ait été ouvert.

On les appelle « plug-ins privés » parce que leur usage est restreint et spécifique à un seul développeur, une application ou bien encore une société. Par conséquent, ils ne sont pas disponibles publiquement.

Les plug-ins publics

Les plug-ins publics sont mis gratuitement à disposition de la communauté qui peut alors les télécharger, les installer et les utiliser librement. C'est d'ailleurs ce qui a été réalisé jusqu'à maintenant puisqu'une partie des fonctionnalités de l'application Jobeet repose sur les deux plug-ins publics `sfDoctrineGuardPlugin` et `sfFormExtraPlugin`.

Les plug-ins publics sont exactement identiques aux plug-ins privés au niveau de leur structure. La seule différence qui les oppose est que n'importe qui peut installer ces plug-ins publics dans ses propres projets. Une partie de ce chapitre se consacre d'ailleurs à présenter comment publier et héberger un plug-in public sur le site officiel de Symfony.

Une autre manière d'organiser le code du projet

Il existe encore une manière supplémentaire de penser aux plug-ins et de les utiliser. Cette fois-ci, il ne s'agit pas de réfléchir en termes de partage et de réutilisation mais en termes d'organisation et d'architecture. En effet, les plug-ins peuvent également être perçus comme une manière totalement différente d'organiser le code d'un projet. Au lieu de structurer les fichiers par couches – tous les modèles dans le répertoire `lib/model` par exemple – les fichiers sont organisés par fonctionnalités. Par exemple, tous les fichiers propres aux offres d'emploi seront regroupés (le modèle, les modules et les templates), tous les fichiers d'un CMS également, etc.

Découvrir la structure de fichiers d'un plug-in Symfony

Plus concrètement, un plug-in est avant tout une architecture de répertoires et de fichiers qui sont organisés d'après une structure prédéfinie, en fonction de la nature des fichiers qu'il contient. L'objectif de ce chapitre est de déplacer la plupart du code de Jobeet écrit jusqu'à présent dans un plug-in dédié `sfJobeetPlugin`. La structure finale de ce plug-in correspondra à celle présentée ci-dessous.

```
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
```

Créer le plug-in sfJobeetPlugin

Il est temps de se consacrer à la création du plug-in `sfJobeetPlugin`. Étant donné que l'application Jobeet contient de nombreux fichiers à déplacer, le processus de création du plug-in se déroulera en sept étapes majeures successives :

1. déplacement des fichiers du modèle ;
2. déplacement des fichiers du contrôleur et de la vue ;
3. déplacement des tâches automatisées ;
4. déplacement des fichiers d'internationalisation de Jobeet ;
5. déplacement du fichier de configuration dédié au routing ;
6. déplacement des fichiers des ressources web ;
7. déplacement des fichiers de l'utilisateur.

Bien entendu, ce processus ne se limite pas seulement à déplacer des fichiers et des dossiers. Certaines parties du code devront être actualisées pour prendre en considération ce changement majeur de l'architecture de Jobeet.

Avant de démarrer le premier point de cette liste d'étapes, le répertoire dédié au plug-in doit être créé dans le projet sous le répertoire `plugins/`.

```
$ mkdir plugins/sfJobeetPlugin
```

CONVENTION

Nommage des noms des plug-ins

Une convention de nommage impose que les noms des plug-ins doivent se terminer par le suffixe `Plugin`. D'autre part, une bonne pratique consiste également à préfixer les noms de plug-ins avec `sf`, bien que ce ne soit pas une obligation.

Migrer les fichiers du modèle vers le plug-in

Déplacer le schéma de description de la base

La première étape du processus de migration vers un plug-in consiste à déplacer tous les fichiers concernant le modèle. C'est l'une des phases les plus critiques car elle nécessite de toucher au code du modèle par endroits, comme il est expliqué plus loin. Pour commencer, il convient de bouger le schéma de description du modèle de Jobeet dans le plug-in.

```
$ mkdir plugins/sfJobeetPlugin/config/
$ mkdir plugins/sfJobeetPlugin/config/doctrine
$ mv config/doctrine/schema.yml plugins/sfJobeetPlugin/config/
doctrine/schema.yml
```

RAPPEL
Manipulation des fichiers d'un projet

Toutes les commandes présentées dans ce chapitre sont relatives aux environnements Unix. Pour les environnements Windows, il suffit de créer manuellement les fichiers, puis de les glisser et de les déposer à partir de l'explorateur de fichiers. Pour les développeurs qui utilisent Subversion ou d'autres outils de gestion du code, il est nécessaire d'avoir recours aux outils que ces logiciels fournissent comme la commande `svn mv` de Subversion pour déplacer les fichiers versionnés d'un dépôt.

Déplacer les classes du modèle, de formulaires et de filtres

Après cela, le plug-in doit accueillir l'ensemble des fichiers du modèle comprenant les classes du modèle, les classes de formulaires ainsi que les classes de filtres. Il faut donc commencer par créer un répertoire `lib/` à la racine du répertoire du plug-in, puis déplacer à l'intérieur les répertoires `lib/model`, `lib/form` et `lib/filter` du projet.

```
$ mkdir plugins/sfJobeetPlugin/lib/
$ mv lib/model/ plugins/sfJobeetPlugin/lib/
$ mv lib/form/ plugins/sfJobeetPlugin/lib/
$ mv lib/filter/ plugins/sfJobeetPlugin/lib/
```

Transformer les classes concrètes en classes abstraites

CONVENTION **Préfixer les noms des classes autogénérées**

Seules les classes qui ont été autogénérées par Doctrine doivent être préfixées avec le mot `Plugin`. Il est strictement inutile de préfixer les classes écrites manuellement.

Après avoir déplacé les classes du modèle, de formulaires, et de filtres, celles-ci doivent être renommées et déclarées abstraites en prenant garde à les préfixer avec le mot `Plugin`. Les exemples qui suivent montrent comment déplacer les nouvelles classes abstraites et les modifier en conséquence.

Voici un exemple de marche à suivre pour déplacer les classes `JobeetAffiliate` et `JobeetAffiliateTable` afin qu'elles deviennent abstraites et puissent être dérivées automatiquement par les nouvelles classes concrètes que Doctrine régénérera à la prochaine reconstruction du modèle.

```
$ mv plugins/sfJobeetPlugin/lib/model/doctrine/
JobeetAffiliate.class.php plugins/sfJobeetPlugin/lib/model/
doctrine/PluginJobeetAffiliate.class.php
```

La définition du nom de la classe `JobeetAffiliate` doit alors être modifiée pour se conformer à celle ci-dessous.

Déclaration de la classe abstraite PluginJobeetAffiliate dans le fichier plugins/sfJobeetPlugin/lib/model/doctrine/PluginJobeetAffiliate.class.php

```
abstract class PluginJobeetAffiliate extends
BaseJobeetAffiliate
{
  public function preValidate($event)
  {
    $object = $event->getInvoker();

    if (!$object->getToken())
    {
      $object->setToken(sha1($object->getEmail().rand(11111, 99999)));
    }
  }

  // ...
}
```

Le processus est à présent exactement le même pour la classe `JobeetAffiliateTable`.

```
$ mv plugins/sfJobeetPlugin/lib/model/doctrine/
JobeetAffiliateTable.class.php plugins/sfJobeetPlugin/lib/
model/doctrine/PluginJobeetAffiliateTable.class.php
```

La classe concrète devient maintenant abstraite et se nomme `PluginJobeetAffiliateTable`.

```
abstract class PluginJobeetAffiliateTable extends
Doctrine_Table
{
  // ...
}
```

Finalement, cette même opération doit être répétée et étendue à toutes les classes du modèle, de formulaires et de filtres. Il suffit tout d'abord de renommer le nom du fichier en prenant en compte le préfixe `Plugin`, et enfin de mettre à jour la définition de la classe afin de la rendre abstraite.

Reconstruire le modèle de données

Une fois que toutes les classes autogénérées ont bien été déplacées, renommées et rendues abstraites, le modèle de données de Jobeet peut alors à son tour être entièrement reconstruit, afin de recréer les classes concrètes qui n'existent plus.

Cependant, avant d'exécuter les tâches automatiques de reconstruction des classes du modèle, de formulaires et de filtres, toutes les classes de base de ces dernières doivent être supprimées du plug-in. Pour ce faire, il suffit simplement d'effacer le répertoire `base/` qui se trouve dans chaque

dossier `plugins/sfJobeetPlugin/lib/*/doctrine` où l'étoile remplace les valeurs `model`, `form` et `filter`.

```
$ rm -rf plugins/sfJobeetPlugin/lib/form/doctrine/base
$ rm -rf plugins/sfJobeetPlugin/lib/filter/doctrine/base
$ rm -rf plugins/sfJobeetPlugin/lib/model/doctrine/base
```

Ce n'est qu'à partir de cet instant que toutes les classes du modèle peuvent être régénérées au niveau du projet en lançant successivement les commandes `doctrine:build-*` où l'étoile prend pour valeur `model`, `forms` et `filters`.

```
$ php symfony doctrine:build-models
$ php symfony doctrine:build-forms
$ php symfony doctrine:build-filters
```

Le résultat de l'exécution de ces trois commandes conduit à la création de nouveaux répertoires au niveau du projet. En effet, le répertoire `lib/model/doctrine` accueille désormais le dossier `sfJobeetPlugin` qui contient lui-même les classes concrètes et les classes de base autogénérées. Celles-ci possèdent maintenant la structure suivante.

- La classe `JobeetJob` hérite des propriétés et méthodes de la classe parente `PluginJobeetJob`, et se situe dans le fichier `lib/model/doctrine/sfJobeetPlugin/JobeetJob.class.php`. `JobeetJob` est la classe la plus spécifique et de plus haut niveau, c'est pour cette raison qu'elle peut accueillir les fonctionnalités spécifiques du projet qui redéfinissent les méthodes prédéfinies du plug-in.
- La classe `PluginJobeetJob` hérite des propriétés et méthodes de la classe parente `BaseJobeetJob`, et se situe dans le fichier `plugins/sfJobeetPlugin/lib/model/doctrine/PluginJobeetJob.class.php`. Cette classe contient l'ensemble des méthodes propres au fonctionnement du plug-in en redéfinissant les méthodes autogénérées par Doctrine.
- La classe `BaseJobeetJob` hérite des propriétés et méthodes de la classe parente `sfDoctrineRecord`, et se situe dans le fichier `lib/model/doctrine/sfJobeetPlugin/base/BaseJobeetJob.class.php`. C'est la classe de base générée à partir du schéma YAML de description de la base de données à chaque fois que la tâche `doctrine:build-model` est exécutée.
- La classe `JobeetJobTable` hérite des propriétés et méthodes de la classe parente `PluginJobeetJobTable`, et se situe dans le fichier `lib/model/doctrine/sfJobeetPlugin/JobeetJobTable.class.php`. `JobeetJobTable` suit exactement le même principe que la classe `JobeetJob` à ceci près que ce sera une instance `Doctrine_Table` qui sera retournée à l'appel de `Doctrine::getTable('JobeetJob')`.

À RETENIR Mettre à jour les classes de formulaires d'un plug-in

Lorsque les classes de formulaires sont déplacées vers le plug-in, il ne faut pas oublier de changer le nom de la méthode `configure()` en `setup()`, puis d'appeler à l'intérieur de celle-ci la méthode `setup()` de la classe parente comme le montre le code ci-après.

```
abstract class
 ➥ PluginJobeetAffiliateForm extends
 ➥ BaseJobeetAffiliateForm
{
  public function setup()
  {
    parent::setup();
  }
  // ...
}
```

À RETENIR Localisation de la classe de base des filtres en Symfony 1.2.0 et 1.2.1

Dans les versions 1.2.0 et 1.2.1 de Symfony, la classe de base des formulaires de filtres se trouve dans le répertoire `plugins/sfJobeetPlugin/lib/filter/base/`. Néanmoins, à l'heure où nous écrivons ces lignes, la version 1.2.5 de Symfony est déjà sortie, c'est pourquoi il ne devrait plus y avoir de raison d'utiliser des versions antérieures à celle-ci.

À RETENIR Risques d'effets indésirables avec un accélérateur PHP

Si un accélérateur PHP tel que APC est installé sur le serveur, il se pourrait que des comportements étranges se produisent après toutes ces modifications. Pour y remédier, il suffit simplement de redémarrer le serveur web Apache.

- La classe `PluginJobeetJobTable` hérite des propriétés et méthodes de la classe parente `Doctrine_Table`, et se situe dans le fichier `lib/model/doctrine/sfJobeetPlugin/PluginJobeetJob.class.php`. Cette classe contient l'ensemble des méthodes propres au fonctionnement du plug-in et l'appel à `Doctrine::getTable('JobeetJob')` retournera une instance de la classe `Doctrine_Table`.

Avec la structure de fichiers ainsi établie, il devient possible de personnaliser les modèles d'un plug-in en éditant la classe de haut niveau `JobeetJob`. De la même manière, le schéma de la base de données peut lui aussi être personnalisé en ajoutant de nouvelles colonnes et relations, et en redéfinissant les méthodes `setTableDefinition()` et `setUp()`.

Supprimer les classes de base des formulaires Doctrine

Maintenant, il faut s'assurer que le plug-in ne contient plus les classes de base des formulaires Doctrine, étant donné que celles-ci sont globales au projet et seront, quoi qu'il en soit, régénérées à l'exécution des tâches automatiques `doctrine:build-forms` et `doctrine:filters`. Si les deux classes `BaseFormDoctrine` et `BaseFormFilterDoctrine` sont présentes dans le plug-in alors elles peuvent être supprimées en toute sécurité.

```
$ rm plugins/sfJobeetPlugin/lib/form/doctrine/
BaseFormDoctrine.class.php
$ rm plugins/sfJobeetPlugin/lib/filter/doctrine/
BaseFormFilterDoctrine.class.php
```

Déplacer la classe Jobeet vers le plug-in

Pour en finir avec le premier point des sept étapes successives à aborder, seule la classe `Jobeet` doit encore être ajoutée dans le plug-in ; il suffit alors de la déplacer depuis le répertoire `lib/` du projet jusque dans le dossier `plugins/sfJobeetPlugin/lib/`.

```
$ mv lib/Jobeet.class.php plugins/sfJobeetPlugin/lib/
```

Ceci étant fait, le cache de Symfony doit être vidé afin de prendre en compte l'ensemble de toutes les modifications apportées ainsi que les nouvelles classes du plug-in. Par la même occasion, il s'avère pertinent de lancer toute la suite de tests unitaires et fonctionnels pour s'assurer que le processus de migration des classes du modèle n'a pas endommagé l'application ou provoqué de régression fonctionnelles.

```
$ php symfony cc
$ php symfony test :all
```

Migrer les contrôleurs et les vues

Cette section aborde la seconde étape du processus de migration d'une application en plug-in. Il y est question du déplacement des modules dans le répertoire du plug-in. Là encore, il s'agit d'une étape décisive et périlleuse dans la mesure où de nombreux changements devront être opérés au niveau du code et des templates. Néanmoins, les tests unitaires et fonctionnels accompagnent le développeur dans ce processus de transition afin de l'aider à déceler les parties du code qui provoquent des erreurs.

Déplacer les modules vers le plug-in

La première étape de cette nouvelle section consiste tout d'abord à créer un répertoire `modules/` dans le plug-in afin d'y déplacer un à un tous les modules de l'application frontend de Jobeet. Cependant un problème se pose : les noms des modules de Jobeet sont bien trop génériques pour pouvoir être embarqués de cette manière dans un plug-in, Ils risqueraient « d'entrer en collision » avec des modules du même nom dans un projet différent. Une bonne pratique est de renommer un à un les modules du plug-in `sfJobeetPlugin` en prenant le soin de tous les préfixer avec `sfJobeet`.

```
$ 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
```

Renommer les noms des classes d'actions et de composants

La modification des noms des modules de Jobeet a un impact immédiat. Tous les noms des classes d'actions (fichiers `actions.class.php`) et de composants (fichiers `components.class.php`) doivent être modifiés car Symfony repose principalement sur des conventions de nommage à respecter pour que l'ensemble reste cohérent et fonctionnel. Ainsi, par exemple, le nom de la classe d'actions du module `sfJobeetAffiliate` devient `sfJobeetAffiliateActions`. Le tableau ci-dessous résume les changements à réaliser pour chaque module.

Tableau 19–1 Liste des modifications à apporter aux classes d'actions et de composants des modules du plug-in

Module	Nom de la classe d'actions	Nom de la classe de composants
sfJobeetAffiliate	sfJobeetAffiliateActions	sfJobeetAffiliateComponents
sfJobeetApi	sfJobeetApiActions	-
sfJobeetCategory	sfJobeetCategoryActions	-
sfJobeetJob	sfJobeetJobActions	-
sfJobeetLanguage	sfJobeetLanguageActions	-

Mettre à jour les actions et les templates

Bien évidemment, il existe encore des références aux anciens noms des modules à la fois dans les templates et dans le corps des méthodes des actions. Il est donc nécessaire de procéder à une mise à jour des noms des modules dans les helpers `include_partial()` et `include_component()` des templates suivants :

- `sfJobeetAffiliate/templates/_form.php` (changer `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`

De la même manière, les actions `search` et `delete` du module `sfJobeetJob` doivent être éditées afin de remplacer toutes les références aux anciens modules dans les méthodes `forward()`, `redirect()` ou `renderPartial()`.

Mise à jour des actions search et delete du module sfJobeetJob dans le fichier plugins/sfJobeetPlugin/modules/sfJobeetJob/actions/actions.class.php

```
class sfJobeetJobActions extends sfActions
{
  public function executeSearch(sfWebRequest $request)
  {
    if (!$query = $request->getParameter('query'))
    {
      return $this->forward('sfJobeetJob', 'index');
    }

    $this->jobs = Doctrine::getTable('JobeetJob') -
>getForLuceneQuery($query);
```

```
    if ($request->isXmlHttpRequest())
    {
      if ('*' == $query || !$this->jobs)
      {
        return $this->renderText('No results.');
      }
      else
      {
        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');
  }

  // ...
}
```

Mettre à jour le fichier de configuration du routage

La modification des noms des modules du plug-in influe nécessairement sur le fichier de configuration `routing.yml` qui contient lui aussi des références aux anciens noms des modules. Par conséquent, l'ensemble des routes déclarées dans ce fichier doivent être éditées.

Contenu du fichier 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 }
```

Activer les modules de l'application frontend

La modification du fichier de configuration `routing.yml` conduit à de nouvelles erreurs. En effet, si l'on tente d'accéder à n'importe quelle page accessible depuis l'une de ces routes en environnement de développement, une exception est automatiquement levée par Symfony indiquant que le module en question n'est pas activé.

Les plug-ins sont partagés par toutes les applications du projet, ce qui signifie aussi que tous les modules sont potentiellement exploitables quelle que soit l'application. Imaginez ce que cela implique en termes de sécurité si un utilisateur parvenait à atteindre le module `sfGuardUser` depuis l'application frontend en découvrant son URL. Cet exemple montre également la raison pour laquelle il est de bonne pratique de supprimer manuellement les routes par défaut de Symfony du fichier de configuration `routing.yml`.

Afin d'éviter ces potentielles failles de sécurité, le framework Symfony désactive par défaut tous les modules pour toutes les applications du projet, et c'est en fin de compte au développeur lui-même de spécifier explicitement dans le fichier de configuration `settings.yml` de l'application, quels sont les modules qu'il souhaite activer ou pas. Cette opération se réalise très simplement en éditant la directive de configuration `enabled_modules` de la section `.settings` du fichier de configuration comme le montre le code ci-dessous.

Activation des modules du plug-in sfJobeetPlugin pour l'application frontend dans le fichier apps/frontend/config/settings.yml

```
all:
  .settings:
    enabled_modules:
      - default
      - sfJobeetAffiliate
      - sfJobeetApi
      - sfJobeetCategory
      - sfJobeetJob
      - sfJobeetLanguage
```

La dernière étape du processus de migration des modules de Jobeet vers le plug-in `sfJobeetPlugin` consiste à corriger les tests fonctionnels qui contrôlent la valeur des modules dans l'objet requête, avant d'exécuter enfin toute la suite des tests en vue de s'assurer que tout est bien rétabli.

```
$ php symfony test:all
```

IMPORTANT Activer des plug-ins dans un projet Symfony

Pour qu'un plug-in soit disponible dans un projet, il doit obligatoirement être activé dans la classe de configuration `ProjectConfiguration` qui se trouve dans le fichier `config/ProjectConfiguration.class.php`. Dans le cas présent, cette étape n'est pas nécessaire, puisque par défaut Symfony agit d'après une stratégie de « liste noire » (*black list*) qui consiste à activer tous les plug-ins à l'exception de ceux qui sont explicitement mentionnés.

```
public function setup()
{
  $this->enableAllPluginsExcept(array('sfDoctrinePlugin',
                                      'sfCompat10Plugin'));
}
```

Cette approche sert à maintenir une compatibilité rétrograde avec d'anciennes versions de Symfony. Néanmoins, il est conseillé de recourir à une approche par « liste blanche » (*white list*) qui consiste à tout désactiver par défaut, puis d'activer les plug-ins au cas par cas comme le prévoit la méthode `enablePlugins()`.

```
public function setup()
{
  $this->enablePlugins(array('sfDoctrinePlugin',
   'sfDoctrineGuardPlugin', 'sfFormExtraPlugin',
   'sfJobeetPlugin'));
}
```

Toutes les étapes critiques du processus de migration de l'application vers un plug-in sont désormais terminées. Les cinq étapes restantes ne sont que formalités, étant donné qu'il ne s'agit principalement que de copier des fichiers existants dans le répertoire du plug-in.

Migrer les tâches automatiques de Jobeet

La migration des tâches automatiques de Jobeet ne pose aucune difficulté puisqu'il s'agit tout simplement de copier le répertoire `lib/task` et ce qu'il contient dans le dossier `plugins/sfJobeetPlugin/lib/`. Une seule ligne de commande permet d'y parvenir.

```
$ mv lib/task plugins/sfJobeetPlugin/lib/
```

Migrer les fichiers d'internationalisation de l'application

De la même manière qu'une application, un plug-in est capable d'embarquer des catalogues de traduction au format XLIFF. La tâche consiste une fois de plus à déplacer le répertoire `apps/frontend/i18n` et tout ce qu'il contient vers la racine du plug-in.

```
$ mv apps/frontend/i18n plugins/sfJobeetPlugin/
```

Migrer le fichier de configuration du routage

La déclaration des règles de routage peut également s'effectuer au niveau du plug-in ; c'est pourquoi dans le cadre de Jobeet, il convient de déplacer le fichier de configuration `routing.yml` dans le répertoire `config/` du plug-in.

```
$ mv apps/frontend/config/routing.yml plugins/sfJobeetPlugin/config/
```

Migrer les ressources Web

Bien que ce ne soit pas toujours très intuitif, un plug-in a aussi la possibilité de contenir un jeu de ressources web comme des images, des feuilles de style en cascade ou bien encore des fichiers JavaScript. Dans la mesure où le plug-in de Jobeet n'a pas vocation à être distribué, il n'est pas nécessaire d'inclure de ressources web. Toutefois, il faut savoir que c'est possible en créant simplement un répertoire `web/` à la racine du plug-in.

Les ressources d'un plug-in doivent être accessibles dans le répertoire `web/` du projet afin d'être atteignables depuis un navigateur web. Le framework Symfony fournit nativement la tâche automatique `plugin:publish-assets` qui se charge de créer les liens symboliques adéquats sur les systèmes Unix, ou de copier les fichiers lorsqu'il s'agit de plates-formes Windows.

```
$ php symfony plugin:publish-assets
```

Migrer les fichiers relatifs à l'utilisateur

Cette dernière étape est un peu plus complexe et cruciale que les autres puisqu'il s'agit de déplacer le code du modèle `myUser` dans une autre classe de modèle dédiée au plug-in. La section suivante explique tout d'abord quels sont les problèmes que l'on rencontre lorsque l'on déplace du code relatif à l'utilisateur, puis en profite pour apporter une solution alternative afin d'y remédier.

Configuration du plug-in

Bien évidemment, il n'est nullement envisageable de copier le fichier de la classe `myUser` directement dans le plug-in étant donné que celle-ci dépend de l'application, et pas particulièrement de Jobeet. Une autre solution éventuelle consisterait à créer une classe `JobeetUser` dans le

plug-in, dans laquelle figurerait tout le code relatif à l'historique des offres de l'utilisateur, et dont la classe `myUser` devrait hériter.

C'est en effet une meilleure approche puisqu'elle isole correctement le code propre au plug-in `sfJobeetPlugin` de celui qui est propre à l'application. Néanmoins, cette solution a ses propres limites puisqu'elle empêche l'objet `myUser` de l'application de recevoir du code métier de la part de plusieurs plug-ins en même temps. C'est exactement le cas ici dans la mesure où l'objet `myUser` est censé accueillir des méthodes en provenance des plug-ins `sfDoctrineGuardPlugin` et `sfJobeetPlugin`, et où malheureusement, PHP ne permet pas l'héritage de classes multiples. Il faut donc résoudre le problème différemment, grâce notamment au nouveau gestionnaire d'événements de Symfony.

Au cours de leur cycle de vie, les objets du noyau de Symfony notifient des événements que l'on peut écouter. Dans le cas de Jobeet, il convient d'écouter l'événement `user.method_not_found`, qui se produit lorsqu'une méthode non définie est appelée sur l'objet `sfUser`.

Lorsque Symfony est initialisé, tous les plug-ins le sont aussi à condition qu'ils disposent d'une classe de configuration similaire à celle ci-dessous.

Contenu de la classe de configuration du plug-in sfJobeetPlugin dans le fichier 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 entièrement gérées par l'objet `sfEventDispatcher`. Enregistrer un écouteur d'événement est simple puisqu'il s'agit d'appeler la méthode `connect()` de cet objet afin de connecter le nom d'un événement à une classe PHP appelable (*PHP callable*).

Développement de la classe JobeetUser

Grâce au code mis en place juste avant, l'objet `myUser` sera capable d'appeler la méthode statique `methodNotFound()` de la classe `JobeetUser` à chaque fois qu'elle sera dans l'impossibilité de trouver une méthode. Il appartient ensuite à la méthode `methodNotFound()` de traiter la fonction manquante ou non.

À RETENIR Notion de classe appelable en PHP (callable)

Une variable PHP appelable (*callable*) est une variable qui peut être utilisée par la fonction `call_user_func()` et qui renvoie `true` lorsqu'elle est testée comme paramètre de la fonction `is_callable()`.

La méthode `is_callable()` accepte en premier argument soit une chaîne de caractères qui contient le nom d'une fonction utilisateur appelable, soit un tableau simple avec deux entrées. Ce tableau peut contenir soit un objet et le nom d'une de ses méthodes publiques, soit le nom d'une classe et une méthode statique publique de celle-ci.

- http://fr.php.net/manual/fr/function.call-user-func.php
- http://fr.php.net/manual/fr/function.is-callable.php

Pour y parvenir, toutes les méthodes de la classe `myUser` doivent être supprimées, avant de créer la nouvelle classe `JobeetUser` suivante dans le répertoire `lib/` du plug-in.

Contenu de la classe myUser du fichier apps/frontend/lib/myUser.class.php

```
class myUser extends sfBasicSecurityUser
{
}
```

Contenu de la classe JobeetUser du fichier 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::getTable('JobeetJob')
        ->createQuery('a')
        ->whereIn('a.id', $ids)
        ->execute();
    } else {
      return array();
    }
  }

  static public function resetJobHistory(sfUser $user)
  {
    $user->getAttributeHolder()->remove('job_history');
  }
}
```

Lorsque le dispatcheur d'événements appelle la méthode statique `methodNotFound()`, il lui passe un objet `sfEvent` en paramètre. Cet objet dispose d'une méthode `getSubject()` qui renvoie le notificateur de l'événement qui, dans le cas présent, correspond à l'objet courant `myUser`.

La méthode `methodNotFound()` teste si la méthode que l'on essaie d'appeler sur l'objet `sfUser` existe dans la classe `JobeetUser` ou pas. Si celle-ci est présente, alors elle est appelée dynamiquement et sa valeur est immédiatement retournée au notificateur par le biais de la méthode `setReturnedValue()` de l'objet `sfEvent`. Dans le cas contraire, Symfony essaiera le tout prochain écouteur enregistré ou lancera une exception.

Avant de tester ces dernières modifications appliquées au projet, il ne faut pas oublier de vider le cache de Symfony puisque de nouvelles classes ont été ajoutées.

```
$ php symfony cc
```

Comparaison des structures des projets et des plug-ins

En résumé, l'approche par plug-in permet à la fois de rendre le code plus modulaire et réutilisable à travers les projets, mais permet aussi d'organiser le code du projet d'une manière plus formelle, puisque chaque fonctionnalité essentielle se trouve isolée à sa place dans le bon plug-in. Le schéma ci-dessous illustre les changements nécessaires pour passer d'une architecture d'un projet à celle d'un plug-in.

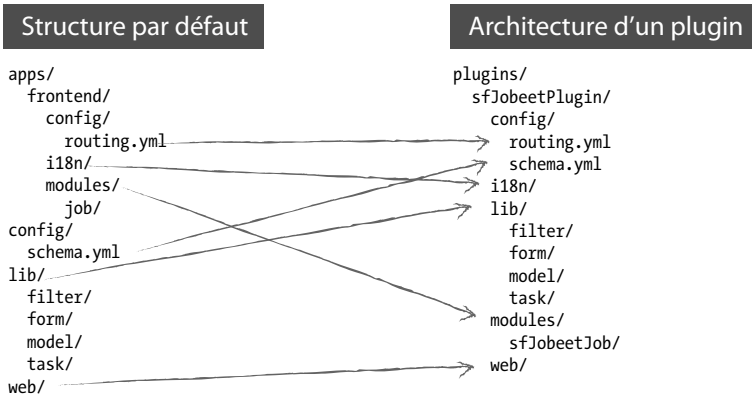


Figure 19–1
Comparaison des architectures des projets et des plug-ins

Utiliser les plug-ins de Symfony

Naviguer dans l'interface dédiée aux plug-ins

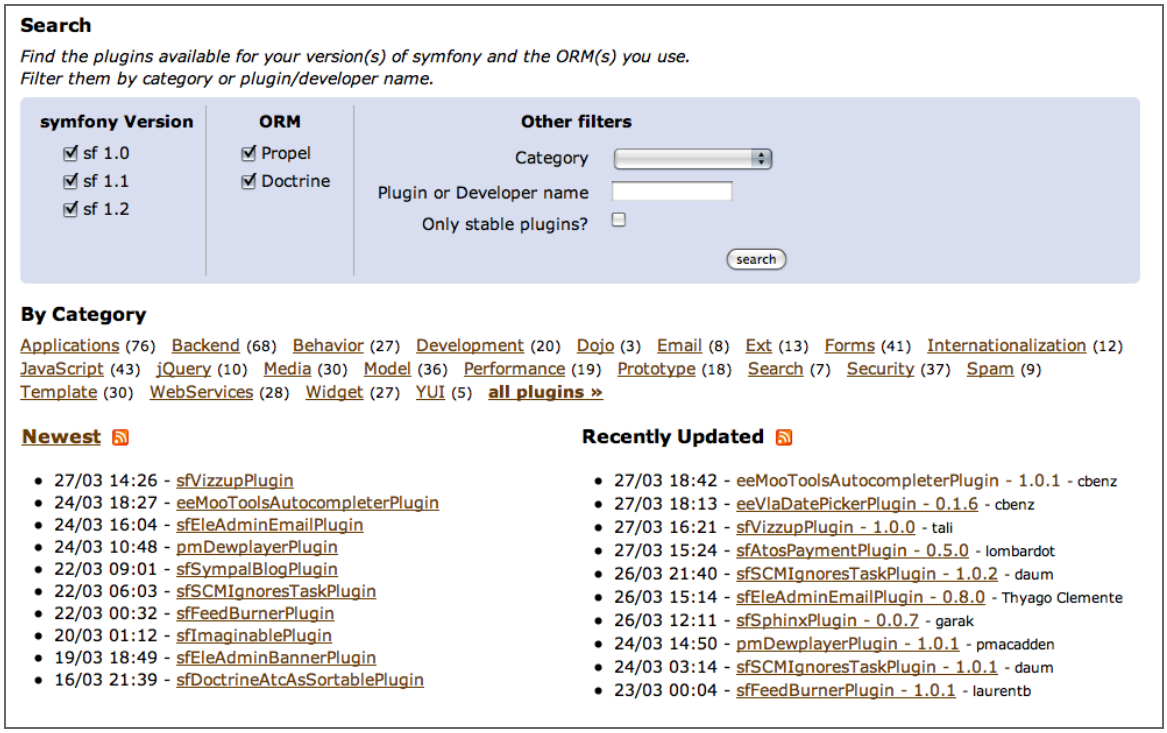


Figure 19–2 Page d'accueil de l'interface des plug-ins du site officiel de Symfony

Lorsqu'il s'agit de démarrer l'implémentation d'une nouvelle fonctionnalité, ou quand un problème récurrent des applications web doit être résolu, il y a de fortes chances que quelqu'un ait déjà rencontré et solutionné ce problème. Dans ce cas, le développeur aura probablement pris le temps de packager sa solution dans un plug-in Symfony.

Le site officiel du projet Symfony dispose d'une section entièrement dédiée aux plug-ins conçus par l'équipe de développement et la communauté, à travers laquelle il est possible de rechercher différentes sortes de plug-ins. Ce dépôt est aujourd'hui riche de plus de 500 plug-ins répartis à travers différentes rubriques comme la sécurité, les formulaires, les e-mails, l'internationalisation, les services web, le JavaScript…

Les différentes manières d'installer des plug-ins

Comme un plug-in est un paquet entièrement autonome compris dans un seul et même répertoire, il existe plusieurs moyens de l'installer. Jusqu'à présent, c'est la méthode la plus courante qui a été employée, c'est-à-dire l'installation grâce à la tâche automatique `plugin:install`.

- Utiliser la tâche automatique `plugin:install`. Cette dernière ne fonctionne qu'à condition que l'auteur du plug-in ait convenablement créé le package du plug-in, puis déposé celui-ci sur le site de Symfony.
- Télécharger le package manuellement et décompresser son contenu dans le répertoire `plugins/` du projet. Cette installation implique également que l'auteur a déposé son package sur le site de Symfony.
- Ajouter le plug-in externe à la propriété `svn:externals` du répertoire `plugins/` du projet à condition que le plug-in soit hébergé sur un dépôt Subversion.

Les deux dernières méthodes sont simples et rapides à mettre en œuvre mais moins flexibles, tandis que la première permet d'installer la toute dernière version disponible du plug-in, en fonction de la version du framework utilisée. De plus, elle facilite la mise à jour d'un plug-in vers sa dernière version stable, et gère aussi aisément les dépendances entre certains plug-ins.

TECHNIQUE La propriété svn:externals de Subversion

La propriété `svn:externals` de Subversion permet d'ajouter à la copie de travail locale, les fichiers d'une ou de plusieurs ressources externes versionnées et maintenues sur un dépôt Subversion distant. En phase de développement, l'utilisation de cette technique a l'avantage de ne pas avoir à gérer soi-même les composants externes utiles à un projet (Swit Mailer par exemple), mais aussi de garder automatiquement la version de ces derniers à jour des dernières modifications. Lors du passage en production finale, la propriété `svn:externals` peut ainsi être éditée pour figer les versions des librairies externes à inclure, afin de ne pas risquer de récupérer des fichiers potentiellement instables de ces dernières par le biais d'un `svn update`.

Contribuer aux plug-ins de Symfony

Packager son propre plug-in

Construire le fichier README

Pour créer le paquet d'un plug-in, il est nécessaire d'ajouter quelques fichiers obligatoires dans la structure interne du plug-in. Le premier, et sans doute le plus important de tous, est le fichier `README` qui est situé à la racine du plugin, et qui explique comment installer le plug-in, ce qu'il fournit mais aussi ce qu'il n'inclut pas.

Le fichier `README` doit également être formaté avec le format Markdown. Ce fichier sera utilisé par le site de Symfony comme principale source de documentation. Une page disponible à l'adresse http://www.symfony-project.org/plugins/markdown_dingus permet de tester les fichiers `README` en convertissant le Markdown en code HTML.

REMARQUE Tâches automatiques de développement de plug-ins

Pour répondre à des besoins fréquents de création de plug-ins privés ou publics, il paraît judicieux de tirer parti de quelques-unes des tâches du plug-in `sfTaskExtraPlugin`. Ce dernier est développé et maintenu par l'équipe de développement de Symfony, et inclut un certain nombre de tâches qui facilitent la rationalisation du cycle de vie des plug-ins.

```
generate:plugin
plugin:package
```

Ajouter le fichier LICENSE

Un plug-in a également besoin de son propre fichier `LICENSE` dans lequel est mentionnée la licence qu'attribue l'auteur à son plug-in. Choisir une licence n'est pas une tâche évidente, mais la section des plug-ins de Symfony liste uniquement ceux qui sont publiés sous une licence similaire à celle du framework (MIT, BSD, LGPL et PHP). Le contenu du fichier `LICENSE` sera affiché dans l'onglet *License* de la page publique du plug-in.

Écrire le fichier package.xml

La dernière étape du processus de création d'un plug-in Symfony consiste en l'écriture du fichier `package.xml` à la racine du plug-in. Ce fichier décrit la composition du plug-in en suivant la syntaxe des paquets PEAR, disponible à l'adresse http://pear.php.net/manual/en/guide-developers.php. Le meilleur moyen d'apprendre à rédiger ce fichier est sans aucun doute de s'appuyer sur le fichier d'un plug-in existant tel que le célèbre `sfGuardPlugin` accessible à l'adresse http://svn.symfony-project.com/plugins/sfGuardPlugin/branches/1.2/package.xml.

Structure générale du fichier package.xml

Contenu du fichier 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>
```

Le noeud contents du fichier package.xml

La balise `<contents>` sert à décrire tous les fichiers et répertoires qui doivent figurer dans le plug-in.

```
<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>
```

Le noeud dependencies du fichier package.xml

Le noeud `<dependencies>` référence toutes les dépendances nécessaires au bon fonctionnement du plug-in comme PHP, Symfony, ou encore les plug-ins. Cette information est utilisée par la tâche automatique `plugin:task` pour installer à la fois la meilleure version du plug-in pour l'environnement du projet, mais aussi pour installer toutes les dépendances obligatoires supplémentaires si elles existent.

```
<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>1.2.0</min>
      <max>1.3.0</max>
      <exclude>1.3.0</exclude>
    </package>
  </required>
</dependencies>
```

Il est recommandé de toujours déclarer la dépendance avec Symfony comme présenté ici. Déclarer une version minimale et maximale permet à la tâche `plugin:install` de savoir quelle version de Symfony est nécessaire étant donné que les versions de Symfony peuvent avoir des APIs légèrement différentes.

Déclarer une dépendance avec un autre plug-in est aussi possible comme le présente le code XML ci-dessous.

```
<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>
```

Le noeud changelog du fichier package.xml

La balise `<changelog>` est optionnelle, mais donne de nombreuses informations utiles à propos de ce qui a changé entre les versions. Cette donnée est affichée dans l'onglet *Changelog* de la page publique du plug-in ainsi que dans le flux RSS dédié de ce celui-ci.

```
<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>
```

Héberger un plug-in public dans le dépôt officiel de Symfony

Héberger un plug-in utile sur le site officiel de Symfony afin de le partager avec la communauté Symfony est extrêmement simple. La première étape consiste à se créer un compte sur le site de Symfony, puis à créer un nouveau plug-in depuis l'interface web dédiée.

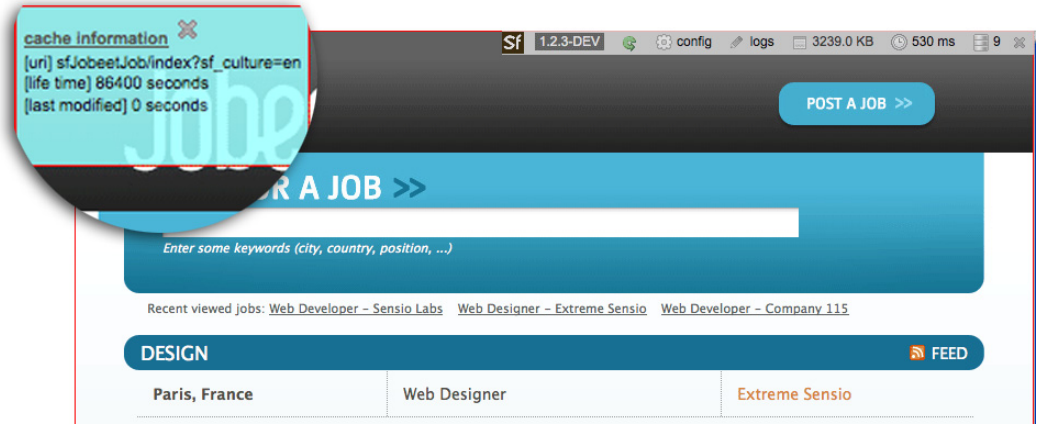
Le rôle d'administrateur du plug-in est automatiquement attribué à l'auteur de la source. Par conséquent, un onglet *admin* est présent dans l'interface. Cet onglet intègre tous les outils et informations nécessaires pour gérer les plug-ins déposés et télécharger les paquets sur le dépôt de Symfony. Une foire aux questions (FAQ) dédiée aux plug-ins est également disponible afin d'apporter un lot d'informations utiles à tous les développeurs de plug-ins.

En résumé…

Créer des plug-ins et les partager avec la communauté est l'une des meilleures façons de contribuer au projet Symfony. C'est si simple à mettre en place que le dépôt officiel des plug-ins de Symfony regorge de plug-ins utiles ou futiles, mais souvent pratiques.

L'application Jobeet arrive doucement à son terme mais il reste encore deux points essentiels à aborder : la gestion du cache de Symfony et le déploiement du projet sur le serveur de production. Ces deux sujets seront respectivement traités dans les deux derniers chapitres de cet ouvrage…

chapitre 20


cache information
[uri] sfJobeetJob/index?sf_culture=en
[life time] 86400 seconds
[last modified] 0 seconds
Sf 1.2.3-DEV config logs 3239.0 KB 530 ms 9
POST A JOB >>
A JOB >>
Enter some keywords (city, country, position, ...)
Recent viewed jobs: Web Developer - Sensio Labs Web Designer - Extreme Sensio Web Developer - Company 115
DESIGN
FEED
Paris, France
Web Designer
Extreme Sensio

La gestion du cache

MOTS-CLÉS :

- Templates, formulaires, partiels, composants
- Environnement de production
- Tests fonctionnels

En environnement de production, une application web se doit d'être fonctionnelle, comme l'établissent ses spécifications, mais aussi performante. Le temps de chargement des pages web est le principal indicateur de performance d'un site Internet et se doit d'être le plus bas possible.

De nombreuses stratégies d'optimisation des pages existent, telle la mise en cache des pages HTML, intégrée nativement dans Symfony.

La question de la performance est plus ou moins commune à toutes les applications web modernes. Le premier critère d'évaluation de performance d'un site Internet est, bien entendu, le calcul du temps de chargement de ses pages. En fonction des résultats obtenus, il convient de procéder ou pas à des travaux d'optimisation en commençant par intégrer au site un cache des pages.

Le framework Symfony intègre nativement plusieurs stratégies de cache en fonction des types de fichier qu'il manipule. Par exemple, les fichiers de configuration YAML ou encore les catalogues de traduction XLIFF sont d'abord convertis en PHP puis sont ensuite mis en cache sur le système de fichier. De plus, le chapitre 12 a montré que les modules générés par le générateur d'administration sont eux aussi cachés pour de meilleures performances.

Ce vingtième et avant dernier chapitre aborde un autre type de cache : le cache de pages HTML. Afin d'améliorer les performances d'une application web, certaines pages du site peuvent être mises en cache.

Pourquoi optimiser le temps de chargement des pages ?

La performance d'une application web constitue un point crucial dans la réussite de celle-ci. En effet, un site Internet se doit d'être accessible en permanence afin de répondre aux besoins des utilisateurs. Or, la satisfaction des exigences des utilisateurs commence à partir du moment où ces derniers se connectent à l'application et attendent que la page se charge complètement.

Le temps que met la page à s'afficher entièrement est un indicateur fondamental puisqu'il permet à l'utilisateur de déterminer s'il souhaite poursuivre ou non sa navigation. Des sites à fort trafic comme Google, Yahoo!, Dailymotion ou encore Amazon ont bien compris la nécessité de consacrer du temps (et de l'argent) à l'optimisation des temps de chargement de leurs pages. Pour étayer ces propos il faut savoir qu'un site comme Amazon perd 1 % de ses ventes si le temps de chargement de ses pages web augmente de 100 ms. Le site Internet de Yahoo!, quant à lui, accuse en moyenne 5 à 9 % d'abandons lorsque ses pages web mettent 400 ms de plus à s'afficher, tandis que Google avoue perdre en moyenne 20 % de fréquentation par demi-seconde de chargement supplémentaire. Ce dernier chiffre explique entre autres l'extrême simplicité de l'interface du moteur de recherche Google et du nombre restreint de liens par page sur leurs pages de résultats[1].

Bien sûr, pour en arriver à un tel point de performance, les équipes de développement de ces sites Internet investissent beaucoup pour mettre en place des stratégies d'optimisation à la fois côté serveur et côté client. Ces chiffres ont pour unique but de démontrer en quoi le temps de chargement des pages web d'une application est déterminant pour sa fréquentation et ses objectifs commerciaux lorsqu'il s'agit d'un site de ventes en ligne par exemple.

Le framework Symfony, qui est par ailleurs utilisé par certains sites de Yahoo! et par Dailymotion, participe à l'élaboration de ce processus d'optimisation en fournissant un système de cache des pages HTML puissant et configurable. Il rend possible un choix d'optimisation au cas par cas en permettant d'ajuster pour chaque page la manière dont elle doit être traitée par le cache.

Créer un nouvel environnement pour tester le cache

Comprendre la configuration par défaut du cache

Par défaut, la fonctionnalité de cache des pages HTML de Symfony est activée uniquement pour l'environnement de production dans le fichier de configuration `settings.yml` de l'application. En revanche, les environnements de développement et de test n'activent pas le cache afin de pouvoir toujours visualiser immédiatement le résultat des pages web tel qu'il sera livré par le serveur au client en environnement de production final.

```
prod:
  .settings:
    cache: on

dev:
  .settings:
    cache: off

test:
  .settings:
    cache: off
```

1. Chiffres recueillis lors d'une conférence d'Éric Daspet au forum PHP 2008 à Paris.

Ajouter un nouvel environnement cache au projet

Configuration générale de l'environnement cache

L'objectif de ce chapitre est d'appréhender et de tester le système de cache des pages de Symfony avant de basculer le projet sur le serveur de production. Cette dernière étape constitue d'ailleurs l'objet du dernier chapitre. Deux solutions sont possibles pour tester le cache : activer le cache pour l'environnement de développement (`dev`) ou créer un nouvel environnement. Il ne faut pas oublier qu'un environnement se définit par son nom (une chaîne de caractères), un contrôleur frontal associé, et optionnellement un jeu de valeurs de configuration spécifiques.

Pour manipuler le système de cache des pages de Jobeet, un nouvel environnement `cache` sera créé et sera similaire à celui de production, à la différence que les logs et les informations de débogage seront disponibles. Il s'agira donc d'un environnement intermédiaire à la croisée des environnements `dev` et `prod`.

Créer le contrôleur frontal du nouvel environnement

La première étape de ce processus de création d'un nouvel environnement pour le cache est de générer à la main un nouveau contrôleur frontal. Pour ce faire, il suffit de copier le fichier `frontend_dev.php`, puis de le copier en le renommant `frontend_cache.php`.

Enfin, la valeur `dev` du second paramètre de la méthode statique `getApplicationConfiguration()` de la classe `ProjectConfiguration` doit être remplacée par `cache`.

Contenu du fichier 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();
```

Le 3e paramètre booléen de la méthode `getApplicationConfiguration()` permet d'activer les informations de débogage comme c'est déjà le cas sur

SÉCURITÉ Empêcher l'exécution des contrôleurs frontaux sensibles

Le code du contrôleur frontal débute par un script qui s'assure que ce dernier est uniquement appelé depuis une adresse IP locale. Cette mesure de sécurité sert à protéger les contrôleurs frontaux sensibles d'être appelés sur le serveur de production. Ce sujet sera détaillé plus en détail au chapitre suivant.

l'environnement de développement. Ce nouvel environnement cache est désormais testable en l'appelant à partir d'un navigateur web par le biais de l'url http://jobeet.localhost/frontend_cache.php/.

Configurer le nouvel environnement

Pour l'instant, l'environnement `cache` hérite de la configuration par défaut de Symfony. Pour lui définir une configuration particulière, il suffit d'éditer le fichier `settings.yml` de l'application frontend en lui ajoutant les paramètres de configuration spécifiques pour l'environnement `cache`.

Configuration de l'environnement cache dans le fichier apps/frontend/config/settings.yml

```
cache:
  .settings:
    error_reporting: <?php echo (E_ALL | E_STRICT)."\n" ?>
    web_debug:       on
    cache:           on
    etag:            off
```

Ces paramètres de configuration activent le niveau de traitement des erreurs le plus fort (`error_reporting`), la barre de débogage de Symfony (`web_debug`) et bien sûr le cache des pages HTML via la directive de configuration `cache`. Comme la configuration par défaut met en cache tous ces paramètres, il est nécessaire de le nettoyer afin de pouvoir constater les changements dans le navigateur.

```
$ php symfony cc
```

Au prochain rafraîchissement de la page dans le navigateur, la barre de débogage de Symfony devrait être présente dans l'angle supérieur droit de la fenêtre, comme c'est déjà le cas en environnement de développement.

Manipuler le cache de l'application

Les sections suivantes entrent véritablement dans le vif du sujet car il s'agit de présenter les différentes stratégies de configuration du cache de Symfony pour l'application Jobeet. En effet, le système de cache natif du framework ne s'arrête pas uniquement à la mise en cache de l'intégralité des pages qu'il génère, mais laisse la possibilité de choisir quelles pages, quelles actions ou encore quels templates doivent être mis en cache.

Configuration globale du cache de l'application

Le mécanisme de cache des pages HTML peut être configuré à l'aide du fichier de configuration `cache.yml` qui se situe dans le répertoire `config/` de l'application. À chaque fois qu'une nouvelle application est générée grâce à la commande `generate:app`, Symfony construit ce fichier et attribue par défaut une configuration minimale du cache.

```
default:
  enabled:     off
  with_layout: false
  lifetime:    86400
```

Par défaut, comme les pages peuvent toutes contenir des informations dynamiques, le cache est désactivé (`enabled: off`) de manière globale pour toute l'application. Il est inutile de modifier ce paramètre de configuration car le cache sera activé ponctuellement page par page dans la suite de ce chapitre. Le paramètre de configuration `lifetime` détermine la durée de vie du cache en secondes sur le serveur. Ici, la valeur 86 400 secondes correspond à une journée complète.

ASTUCE **Configurer le cache différemment**

Une autre manière de gérer la configuration du cache est également possible. Il s'agit de faire l'inverse, c'est-à-dire d'activer le cache globalement puis de le désactiver ponctuellement pour les pages qui n'ont pas besoin d'être mises en cache. Finalement, tout dépend de ce qui représente le moins de travail pour l'application et pour le développeur.

Activer le cache ponctuellement page par page

Activation du cache de la page d'accueil de Jobeet

Comme la page d'accueil de Jobeet sera sans doute la plus visitée de tout le site Internet, il est particulièrement pertinent de la mettre en cache afin d'éviter d'interroger la base de données à chaque fois que l'utilisateur y accède. En effet, il est inutile de reconstruire complètement la page si celle-ci a déjà été générée une première fois quelques instants auparavant. Pour forcer la mise en cache de la page d'accueil de Jobeet, il suffit de créer un fichier `cache.yml` pour le module `sfJobeetJob`.

ASTUCE **Le fichier cache.yml**

Le fichier de configuration `cache.yml` possède les mêmes propriétés que tous les autres fichiers de configuration tels que `view.yml`. Cela signifie par exemple que le cache peut être activé pour toutes les actions d'un même module en utilisant la section spéciale `all`.

Configuration du cache de la page d'accueil de Jobeet dans le fichier plugins/sfJobeetJob/modules/sfJobeetJob/config/cache.yml

```
index:
  enabled:     on
  with_layout: true
```

En rafraîchissant le navigateur, on constate que Symfony a décoré la page avec une boîte bleue indiquant que le contenu a été mis en cache sur le serveur.

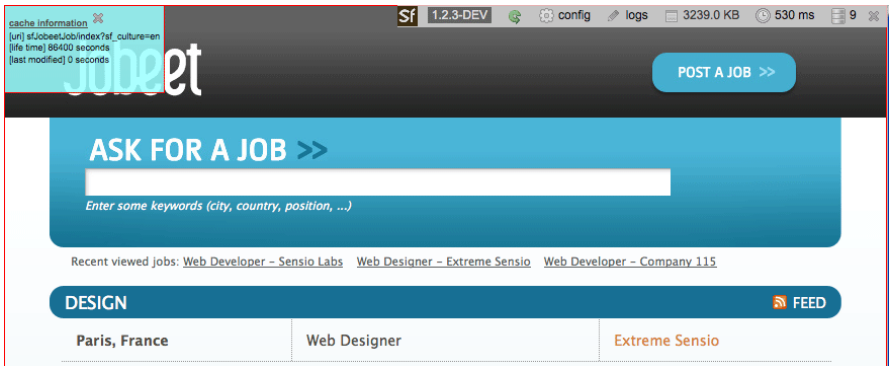


Figure 20–1 Boîte d'information du cache de la page après sa mise en cache

Cette boîte donne de précieuses informations pour le débogage concernant la clé unique du cache, sa durée de vie ou encore son âge. En rafraîchissant la page une nouvelle fois dans le navigateur, la boîte d'information du cache passe du bleu au jaune, ce qui indique que la page a été directement retrouvée dans le cache du serveur de fichiers.

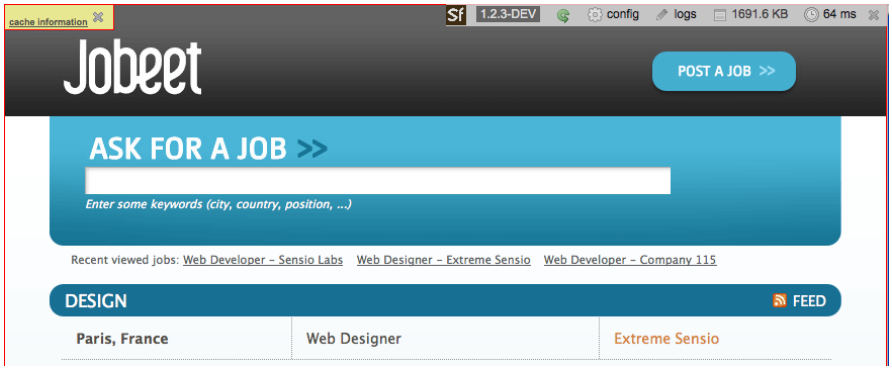


Figure 20–2 Boîte d'information du cache de la page après sa récupération depuis le cache

Il faut aussi remarquer qu'aucune requête SQL à la base de données n'a été réalisée dans cette seconde génération de la page d'accueil. La barre de débogage de Symfony est présente pour en témoigner.

Principe de fonctionnement du cache de Symfony

Lorsqu'une page est « cachable » et que le fichier de cache de cette dernière n'existe pas encore sur le serveur, Symfony se charge de sauvegarder automatiquement l'objet de la réponse dans le cache à la fin de la requête. Pour toutes les requêtes suivantes, Symfony renverra la réponse déjà cachée sans avoir à appeler le contrôleur. Le schéma ci-dessous illustre le cycle de fonctionnement d'une page destinée à une mise en cache.

À RETENIR Mise en cache de la page d'accueil en fonction de la langue

Même si la langue peut être modifiée simplement par l'utilisateur, le cache continuera de fonctionner, étant donné que celle-ci est directement embarquée dans l'URL.

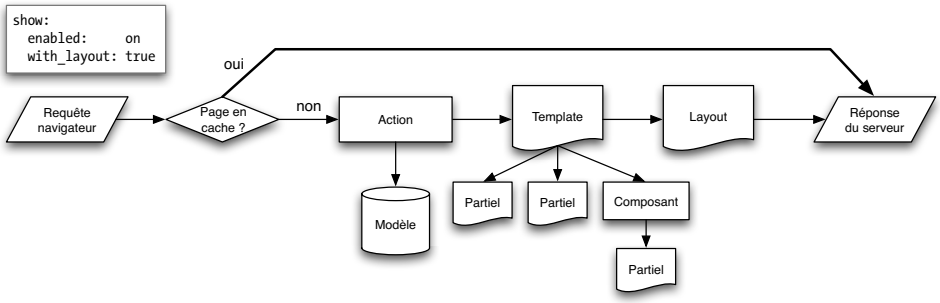


Figure 20–3
Schéma de description du cycle de mise en cache d'une page

À RETENIR Fonctionnement du cache avec les méthodes HTTP

Une requête entrante contenant des paramètres GET ou soumise à partir des méthodes POST, PUT ou DELETE, ne sera jamais mise en cache par Symfony, quelle que soit la configuration.

Ce système a un impact positif immédiat sur les performances de la page. Pour s'en convaincre, des outils Open Source comme le célèbre `JMeter` (http://jakarta.apache.org/jmeter/) pour Apache permettent de mesurer les temps de réponse de chaque requête afin de générer des statistiques.

Activer le cache de la page de création d'une nouvelle offre

Au même titre que la page d'accueil de Jobeet, la page de création d'une nouvelle offre d'emploi du module `sfJobeetJob` peut être mise en cache en spécifiant la configuration suivante dans le fichier `cache.yml`.

Configuration de la mise en cache de la page de création d'une offre dans le fichier plugins/sfJobeetJob/modules/sfJobeetJob/config/cache.yml

```
new:
  enabled:     on

index:
  enabled:     on

all:
  with_layout: true
```

Comme les deux pages peuvent être mises en cache avec le layout, la directive de configuration `with_layout` a été mutualisée dans la section `all` afin qu'elle s'applique à toutes les pages du module `sfJobeetJob`.

Nettoyer le cache de fichiers

Pour nettoyer tout le cache des pages HTML, il suffit d'exécuter la commande `cache:clear` de Symfony comme cela a déjà été utilisé à maintes reprises tout au long de cet ouvrage.

```
$ php symfony cc
```

La tâche automatique `cache:clear` supprime tous les fichiers de cache de Symfony qui se trouvent dans le répertoire `cache/` du projet. Pour éviter de supprimer l'intégralité du cache, cette commande est accompagnée de quelques options qui lui permettent de supprimer sélectivement certaines parties du cache. Par exemple, pour ne supprimer que les templates mis en cache pour l'environnement `cache`, il existe les deux options `--type` et `--env`.

```
$ php symfony cc --type=template --env=cache
```

Au lieu de nettoyer le cache à chaque fois qu'un changement est réalisé, il est possible d'annuler la mise en cache en ajoutant simplement une chaîne de requête dans l'URL, ou en utilisant le bouton *Ignore cache* de la barre de débogage de Symfony.

Figure 20–4 Gestion du cache depuis la barre de déboguage de Symfony

Activer le cache uniquement pour le résultat d'une action

Exclure la mise en cache du layout

Il arrive parfois qu'il soit impossible de mettre en cache la page entière, alors que le template évalué d'une action peut lui-même être mise en cache. Dit d'une autre manière, il s'agit en fait de tout cacher à l'exception du layout.

Pour l'application Jobeet, le cache des pages entières est impossible en raison de la présence de la barre d'historique de consultation des dernières offres d'emploi de l'utilisateur. Cette dernière varie en effet constamment de page en page. Par conséquent la paramétrage du cache doit être mis à jour en désactivant la directive de configuration propre au layout.

Suppression de la mise en cache du layout pour toutes les actions du module sfJobeetJob dans le fichier plugins/sfJobeetJob/modules/sfJobeetJob/config/cache.yml

```
new:
  enabled:     on

index:
  enabled:     on

all:
  with_layout: false
```

La désactivation de la mise en cache du layout dans la directive de configuration `with_layout` nécessite de réinitialiser tout le cache de Symfony.

```
$ php symfony cc
```

L'actualisation de la page dans le navigateur suffit alors pour constater la différence.

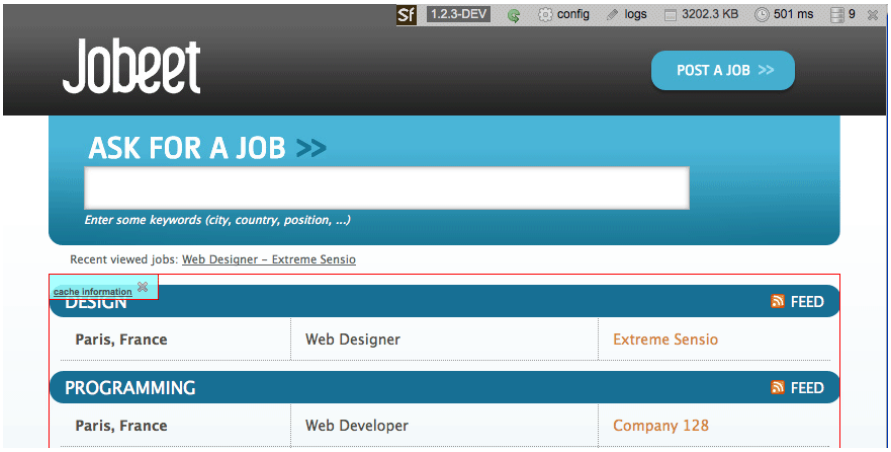


Figure 20–5 Mise en cache du résultat d'une action sans le layout

Fonctionnement de la mise en cache sans layout

Bien que le flux de la requête soit à peu près similaire à celui de ce schéma simplifié, il n'en demeure pas moins que la mise en cache des pages sans layout consomme davantage de ressources. En effet, la page cachée n'est plus renvoyée immédiatement si elle existe car elle doit d'abord être décorée par le layout, ce qui génère une perte notable de performance par rapport à la mise en cache globale.

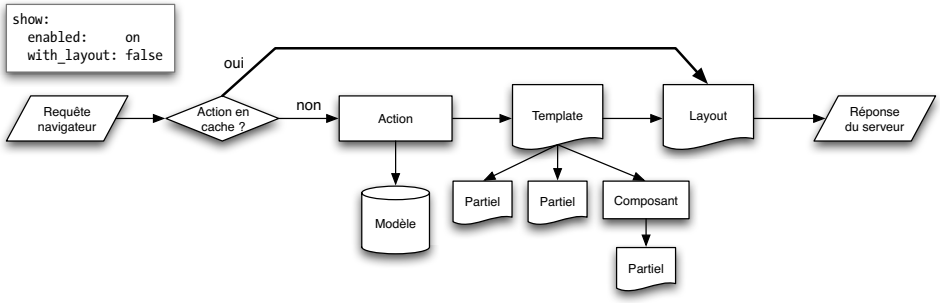


Figure 20–6 Cycle de fonctionnement d'une page cachée sans son layout

Activer le cache des templates partiels et des composants

Configuration du cache

Il est bien souvent impossible de cacher la globalité du résultat d'un template évalué par une action pour les sites hautement dynamiques. En effet, les templates d'action peuvent embarquer des entités dynamiques supplémentaires telles que les templates partiels ou les composants qui réagissent différemment en fonction de l'utilisateur courant ou des paramètres qui leur sont affectés.

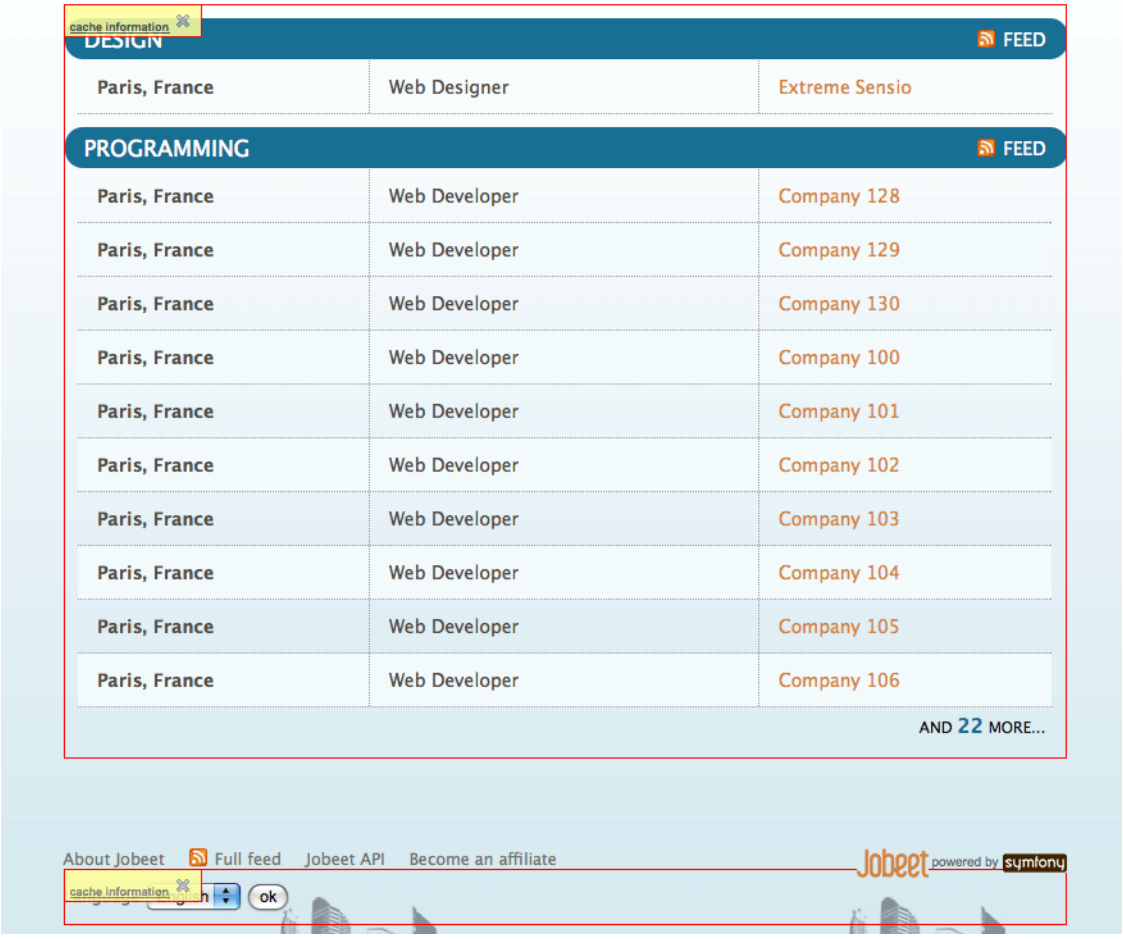


Figure 20–7 Mise en cache des templates partiels et des composants

Par conséquent, il est nécessaire de trouver un moyen de configurer le cache au niveau de granularité le plus fin. Fort heureusement dans Symfony, les templates partiels et les composants peuvent être cachés de manière indépendante. La capture d'écran ci-dessus témoigne de la mise

en cache du template partiel `_list.php` qui génère la liste dynamique des offres d'emploi, ainsi que du composant de changement manuel de la langue du site dans le pied de page du layout.

Il convient donc de configurer la mise en cache du composant de changement de langue en créant un nouveau fichier `cache.yml` dans le module `sfJobeetLanguage`. Le code ci-dessous en présente le contenu.

Configuration du cache pour le composant language dans le fichier plugins/sfJobeetJob/modules/sfJobeetLanguage/config/cache.yml

```
_language:
  enabled: on
```

À RETENIR Désactivation automatique de la mise en cache du layout

Configurer le cache pour un template partiel ou un composant est aussi simple que d'ajouter une nouvelle entrée du même nom que le template au fichier `cache.yml`. Pour ces types de fichier, l'option `with_layout` est volontairement supprimée et non prise en compte par Symfony ; cela n'aurait en effet aucun sens.

Principe de fonctionnement de la mise en cache

Le nouveau schéma ci-dessous décrit la stratégie de mise en cache des templates partiels et des composants dans le flux de la requête.

IMPORTANT Contextuel ou non ?

Le même composant ou template partiel peut être appelé par différents templates. C'est le cas par exemple du partiel `_list.php` de génération de la liste des offres d'emploi qui est utilisé à la fois dans les modules `sfJobeetJob` et `sfJobeetCategory`. Comme le rendu de ce template est toujours le même, le partiel ne dépend donc pas du contexte dans lequel il est utilisé et le cache reste le même pour tous les templates (le cache d'un fichier est évidemment systématiquement unique pour un jeu de paramètres différents).
Néanmoins, il arrive parfois que le rendu d'un partiel ou d'un composant soit différent, en fonction de l'action qui l'inclut (dans le cas, par exemple, de la barre latérale d'un blog qui est légèrement différente pour la page d'accueil et pour la page d'un billet). Dans ces cas-là, le partiel ou le composant est contextuel, et le cache doit être configuré en conséquence en paramétrant l'option `contextual` à la valeur `true`.

```
_sidebar:
  enabled:    on
  contextual: true
```

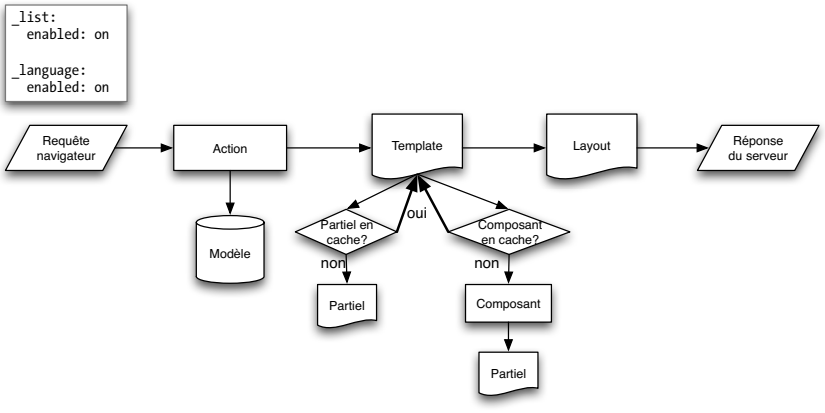


Figure 20–8
Schéma de description de la mise en cache des partiels et des composants

Activer le cache des formulaires

Comprendre la problématique de la mise en cache des formulaires

Sauvegarder la page de création d'une nouvelle offre dans le cache pose un véritable problème étant donné que cette dernière contient un formulaire. Pour mieux comprendre de quoi il s'agit, il est nécessaire de se rendre sur la page *Post a Job* avec le navigateur pour générer et mettre la page en cache. Une fois cette étape réalisée, les cookies de session du navigateur doivent être réinitialisés, et la page rafraîchie. À présent, la tentative de soumission du formulaire provoque une erreur globale alertant d'une éventuelle attaque CSRF.

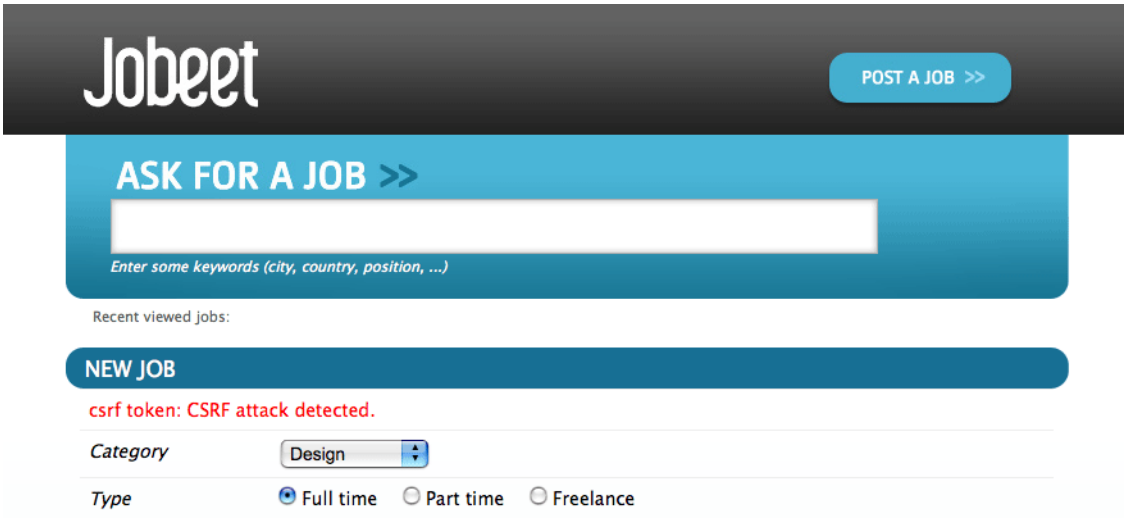


Figure 20–9 Résultat d'une attaque CSRF dans le formulaire de création d'une nouvelle offre

Que s'est-il passé exactement ? La réponse est simple. En effet, à la création de l'application dans les premiers chapitres, un mot de passe CSRF a été défini pour sécuriser l'application. Symfony se sert de ce mot de passe pour générer et embarquer un jeton unique dans tous les formulaires. Ce jeton est généré pour chaque utilisateur et chaque formulaire, et protège l'application d'une éventuelle attaque CSRF.

La première fois que la page est affichée, le code HTML du formulaire est généré et stocké dans le cache avec le jeton unique de l'utilisateur courant. Si un nouvel utilisateur arrive juste après avec son propre jeton, ce sera malgré tout le résultat caché avec le jeton du premier utilisateur qui sera affiché. À la soumission du formulaire, les deux jetons ne correspondent pas et la classe du formulaire lance une erreur.

Désactiver la création du jeton unique

Comment ce problème peut-il être résolu, alors qu'il est légitime de vouloir sauvegarder le formulaire dans le cache ? Le formulaire de création d'une nouvelle offre ne dépend pas de l'utilisateur et ne change absolument rien pour l'utilisateur courant. Dans ce cas, aucune protection CSRF n'est nécessaire et la génération du jeton peut tout à fait être supprimée.

Suppression du jeton CSRF du formulaire de création d'une nouvelle offre dans le fichier plugins/sfJobeetPlugin/lib/form/doctrine/PluginJobeetJobForm.class.php

```
abstract PluginJobeetJobForm extends BaseJobeetJobForm
{
  public function __construct(sfDoctrineRecord $object = null,
$options = array(), $CSRFSecret = null)
  {
    parent::__construct($object, $options, false);
  }

  // ...
}
```

Il ne reste alors plus qu'à vider le cache et réessayer le scénario précédent afin de prouver que tout fonctionne correctement. La même configuration doit également être étendue au formulaire de modification de la langue du site, puisque celui-ci est contenu dans le layout et sera sauvé dans le cache. Comme la classe par défaut `sfLanguageForm` est utilisée et qu'il n'est pas nécessaire de créer une nouvelle classe, la désactivation du jeton CSRF peut être réalisée depuis l'extérieur de la classe dans l'action et le composant du module `sfJobeetLanguage`.

Désactivation du jeton CSRF depuis le composant language du fichier plugins/sfJobeetJob/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')));
    unset($this->form[$this->form->getCSRFFieldName()]);
  }
}
```

Désactivation du jeton CSRF depuis l'action changeLanguage du fichier plugins/sfJobeetJob/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')));
    unset($form[$form->getCSRFFieldName()]);

    // ...
  }
}
```

La méthode `getCSRFFieldName()` retourne le nom du champ qui contient le jeton CSRF. En supprimant ce champ, le widget et le validateur associés sont automatiquement retirés du processus de génération et de contrôle du formulaire.

Retirer le cache automatiquement

Configurer la durée de vie du cache de la page d'accueil

Chaque fois que l'utilisateur poste et active une nouvelle offre, la page d'accueil de Jobeet doit être rafraîchie pour afficher la nouvelle annonce dans la liste. Néanmoins, il n'est pas urgent de la faire apparaître en temps réel sur la page d'accueil, c'est pourquoi la meilleure stratégie consiste à diminuer la durée de vie du cache à une période acceptable.

Configuration de la durée de vie du cache de la page d'accueil dans le fichier plugins/sfJobeetJob/modules/sfJobeetJob/config/cache.yml

```
index:
  enabled: on
  lifetime: 600
```

Au lieu d'affecter la durée de vie par défaut d'une journée à cette page, le cache sera automatiquement supprimé et régénérer toutes les dix minutes.

Forcer la régénération du cache depuis une action

Toutefois, s'il est véritablement nécessaire de mettre à jour la page d'accueil dès que l'utilisateur active sa nouvelle offre, il faut alors modifier l'action `executePublish()` du module `sfJobeetJob` afin de forcer manuellement le cache.

Régénération manuelle du cache depuis une action dans le fichier plugins/sfJobeetJob/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 des pages HTML est exclusivement géré par la classe `sfViewCacheManager`. La méthode `remove()` supprime le cache associé à une URL interne. Pour retirer le cache pour toutes les valeurs possibles d'une variable, il suffit de spécifier le caractère étoile `*` comme valeur. Ainsi, l'utilisation de la chaîne `sf_culture=*` dans le code ci-dessus signifie que Symfony supprimera le cache pour les pages d'accueil de Jobeet en français et en anglais.

Comme le gestionnaire de cache est nul lorsque le cache est désactivé, la suppression du cache a été encapsulée dans un bloc conditionnel.

AVERTISSEMENT **La classe sfContext**

L'objet `sfContext` contient toutes les références aux objets du noyau de Symfony comme la requête, la réponse, l'utilisateur, etc. Puisque la classe `sfContext` agit comme un singleton, il est possible d'avoir recours à l'instruction `sfContext::getInstance()` pour le récupérer depuis n'importe où et ensuite avoir accès à tous les objets métiers du noyau.

```
$user = sfContext::getInstance()->getUser();
```

Toutefois, il est bon de réfléchir à deux fois avant de faire appel à `sfContext::getInstance()`dans les autres classes car cela engendre un couplage fort. Il est toujours préférable de passer l'objet `sfContext` en paramètre d'une classe qui en a besoin.

Il est également possible d'utiliser `sfContext` comme un registre pour y stocker des objets en utilisant la méthode `set()`, qui prend un nom et un objet en guise de paramètres. La méthode `get()`, quant à elle, permet de récupérer un objet du registre par le biais de son nom.

```
sfContext::getInstance()->set('job', $job);
$job = sfContext::getInstance()->get('job');
```

Tester le cache à partir des tests fonctionnels

Activer le cache pour l'environnement de test

Avant de démarrer l'écriture de nouveaux tests fonctionnels, la configuration de l'environnement de test doit être mise à jour afin d'activer la couche de cache des pages HTML. Le code ci-dessous indique les modifications à réaliser dans le fichier de configuration `settings.yml` de l'application frontend.

Activation du cache pour l'environnement de test dans le fichier apps/frontend/config/settings.yml

```
test:
  .settings:
    error_reporting: <?php echo ((E_ALL | E_STRICT) ^
E_NOTICE)."\n" ?>
    cache:           on
    web_debug:       off
    etag:            off
```

Tester la mise en cache du formulaire de création d'une offre d'emploi

La dernière étape consiste à mettre à jour la suite de tests fonctionnels afin de vérifier que les pages de Jobeet configurées tout au long de ce chapitre sont correctement mises en cache par le serveur. Puisqu'il a beaucoup été question de la page de création d'une nouvelle annonce, les tests suivants contrôlent que cette dernière est bien cachée automatiquement.

Test de la mise en cache de la page de création d'une nouvelle offre dans le fichier test/functional/frontend/jobActionsTest.php

```
$browser->
  info(' 7 - Job creation page')->

  get('/fr/')->
  with('view_cache')->isCached(true, false)->

  createJob(array('category_id' =>
Doctrine::getTable('CategoryTranslation')
  ->findOneBySlug('programming')->getId()), true)->

  get('/fr/')->
  with('view_cache')->isCached(true, false)->
  with('response')->checkElement('.category_programming
.more_jobs', '/23/')
;
```

ASTUCE Tester l'environnement de test dans un navigateur

Même avec tous les outils fournis par le framework de tests fonctionnels, il est parfois plus facile de diagnostiquer des problèmes à travers le navigateur. C'est vraiment très simple à mettre en œuvre puisqu'il suffit de créer un contrôleur frontal pour l'environnement de test. Les traces de logs sauvegardés dans le fichier `log/frontend_test.log` apportent de nombreuses informations utiles pour le débogage.

Le testeur `view_cache` est employé ici pour vérifier ce qui figure dans le cache par le biais de la méthode `isCached()` qui accepte deux valeurs booléennes en paramètres :

- la première pour déterminer si oui ou non la page doit être dans le cache ;
- la seconde pour déterminer si oui ou non la page cachée comprend le layout.

En résumé…

Comme la plupart des autres fonctionnalités de Symfony, le sous-framework natif de cache est particulièrement flexible puisqu'il permet au développeur de configurer le cache des fichiers au niveau de granularité le plus fin. Cette gestion poussée du cache des pages HTML offre l'avantage de personnaliser les stratégies de mise en cache aussi bien de manière globale pour un ensemble de fichiers et d'actions, que de manière individualisée.

Le prochain chapitre achève la réalisation de cette étude de cas Jobeet en s'intéressant à la toute dernière étape du développement d'une application Internet : le déploiement en production. Cette étape est particulièrement décisive et mérite toutes les attentions de la part du développeur. Le chapitre traitera de configuration, d'initialisation et de sécurisation de l'application, mais également de la personnalisation des pages d'erreur 404 et 500, de l'activation d'un cache d'opcodes et bien évidemment du déploiement des fichiers source sur le serveur de production.


symfony

chapitre 21

Le déploiement en production

MOTS-CLÉS :

- Accélérateur PHP APC
- Personnalisation des erreurs 404
- Déploiement via rsync

Après les nombreuses heures passées en développement, puis en environnement de recette, la dernière étape décisive de création d'une application web est bien évidemment le déploiement sur le serveur de production final. Cette étape est cruciale car elle implique de paramétrer soigneusement le serveur et le projet Symfony afin d'éviter toute faille de sécurité ou dysfonctionnement.

Heureusement, le framework Symfony dispose d'outils qui facilitent ces phases de déploiement critiques.

Le chapitre précédent a été l'occasion de préparer le terrain en configurant pas à pas le cache des pages HTML de l'application Jobeet pour l'environnement de production. Par conséquent, la dernière étape du projet consiste à déployer les fichiers source vers le serveur de production final.

Tout au long de ces vingt et un chapitres, l'application a été construite sur une machine de développement, probablement un serveur web local personnel pour la plupart des développeurs… À présent, le moment est venu de déplacer tout le site sur le serveur de production afin de le rendre disponible aux internautes. Cependant, la mise en production d'une application n'est jamais aussi simple qu'elle n'y paraît. Il faut en effet penser à préparer et paramétrer plusieurs composants, à commencer par le serveur web. Ce dernier chapitre se consacre donc entièrement à ce sujet afin de présenter quelles sortes de stratégies de déploiement employer, quelles méthodologies suivre ou bien encore quels outils mettre en place pour faciliter et réussir un déploiement…

Préparer le serveur de production

Le premier élément clé du processus de déploiement d'un projet est bien sûr le serveur web car c'est lui qui accueille l'application. Son installation et par extension, sa configuration, doivent le rendre capable d'interpréter les fichiers source du site web. Par conséquent, la machine doit au minimum disposer d'un serveur web (Apache 2), d'un serveur de bases de données (MySQL) et d'une version de PHP récente, c'est-à-dire strictement supérieure ou égale à la 5.2.4. Ces trois outils forment le tiercé de base d'une plate-forme LAMP destinée à recevoir une application web.

L'installation de ces logiciels n'étant pas le sujet principal de cet ouvrage, la suite du chapitre considère qu'ils sont correctement installés sur le serveur de production. Quant à la configuration du serveur web Apache, elle a été largement traitée au premier chapitre.

INFORMATION

Accès SSH en ligne de commande

La majorité de ce chapitre fait usage de la ligne de commande par le biais d'un tunnel SSH entre la machine locale et le serveur web distant. Si votre serveur web ne supporte pas les connexions SSH, ou plus généralement si l'accès en ligne de commande est interdit, alors vous pouvez sauter les sections pour lesquelles ces outils sont nécessaires.

Vérifier la configuration du serveur web

Tout d'abord, il est primordial de vérifier que PHP est installé sur le serveur web avec toutes les extensions nécessaires, et qu'il est bien sûr configuré convenablement (`magic_quotes_gpc` à `off`, `register_globals` à `off`…). De la même manière qu'au tout premier chapitre de cet ouvrage, c'est le script `check_configuration.php` fourni par Symfony qui sera employé pour contrôler la configuration globale du serveur. Étant donné que Symfony n'est pas encore installé par défaut sur la machine de pro-

duction, ce script PHP doit d'abord être récupéré depuis les sources du framework, et ensuite installé sur le serveur web.

```
http://trac.symfony-project.org/browser/branches/1.2/data/bin/
  ➥ check_configuration.php?format=raw
```

Il suffit alors de copier et de coller le fichier à la racine du serveur de production, puis de l'exécuter depuis un navigateur, mais également en ligne de commande afin de s'assurer que les deux environnements sont configurés de la même manière pour Symfony.

```
$ php check_configuration.php
```

L'objectif de ce fichier est d'aider à déceler les défauts de configuration de la plate-forme PHP ou les extensions obligatoires manquantes comme les drivers PDO, le support du XML ou encore les fonctions de conversion d'encodage. À l'exécution de ce script, si des erreurs sont révélées, il est important de les corriger une par une, puis de relancer le script obligatoirement dans les deux environnements.

Installer l'accélérateur PHP APC

Pour le serveur de production, il est évident qu'il faut obtenir les meilleures performances possibles, et l'installation d'un accélérateur PHP y contribue pour beaucoup en apportant les meilleures améliorations gratuitement. Pour le langage PHP, de nombreuses solutions Open Source éprouvées existent comme APC, eAccelerator ou encore Turck MMCache. La société Zend propose quant à elle une solution payante toute aussi performante. Comme APC est l'une des solutions les plus populaires et qu'il sera aussi livré par défaut à la sortie de PHP 6, c'est lui qui a été choisi pour améliorer les performances de PHP. Son installation est particulièrement simple puisqu'elle se résume à l'exécution d'une seule commande.

```
$ pecl install APC
```

L'installation d'APC dépend du système d'exploitation installé, c'est pourquoi il n'est pas à exclure que des dépendances supplémentaires pour ce dernier puissent être installées.

TECHNOLOGIE Comment fonctionne un accélérateur PHP ?

Un accélérateur PHP est un logiciel qui se charge de mettre en cache le code précompilé, l'opcode et des scripts PHP afin d'éviter le temps supplémentaire d'interprétation et de précompilation du code source à chaque requête. Les performances sont au rendez-vous puisque l'on constate généralement un gain de l'ordre de 100 % voire plus, soit plus du double des performances sans accélérateur PHP. Pour en savoir plus au sujet d'APC, la documentation officielle du projet se trouve à l'adresse http://www.php.net/manual/en/apc.configuration.php.

Installer les librairies du framework Symfony

Embarquer le framework Symfony

L'une des forces majeures de Symfony est la complète autonomie du projet. En effet, tous les fichiers nécessaires au bon fonctionnement du projet se trouvent à la racine du répertoire principal de ce dernier. Par

conséquent, le projet peut être déplacé facilement d'un répertoire à un autre sans que l'on ait à modifier quoique ce soit, grâce notamment à l'utilisation de chemins relatifs par Symfony. Cela signifie aussi que le répertoire du serveur de production ne doit pas être nécessairement le même que celui qui se trouve sur la machine de développement.

Le seul chemin absolu potentiel que l'on peut trouver figure dans le fichier `config/ProjectConfiguration.class.php` ; mais le chapitre 1 a permis d'y faire attention et de le remplacer par un chemin relatif. Néanmoins, si ce n'était pas encore le cas, ce chemin absolu vers le composant interne de chargement automatique des classes de Symfony peut être transformé en chemin relatif de la manière suivante.

Transformation du chemin absolu vers la classe sfCodeAutoload en chemin relatif dans le fichier config/ProjectConfiguration.class.php

```
require_once dirname(__FILE__).'/../lib/vendor/symfony/lib/
autoload/sfCoreAutoload.class.php';
```

Garder Symfony à jour en temps réel

Bien que le projet et le framework soient entièrement autonomes, il est aisé de maintenir Symfony à jour à la sortie de nouvelles versions mineures.

Malgré tout le soin apporté au développement du framework, l'équipe de Symfony corrige continuellement des bogues et parfois des failles de sécurité avant de les publier à l'occasion de la sortie d'une nouvelle version mineure. Par conséquent, il est possible de garder la version de Symfony à jour en temps réel pour n'importe quelle application en production.

L'avantage avec le projet Symfony, c'est que toutes les versions majeures (1.0, 1.1 et 1.2) du framework sont maintenues au moins un an, et pendant ces périodes de maintenance, aucune nouvelle fonctionnalité n'est ajoutée mais uniquement des correctifs. Cette approche permet ainsi de conserver une stabilité du framework et de toujours garantir un processus de mise à jour rapide, sécurisé et sans risque d'une version mineure à une autre.

Mise à jour manuelle de Symfony depuis l'archive ZIP

Mettre à jour Symfony est extrêmement simple puisqu'il suffit de modifier le contenu du répertoire `lib/vendor/symfony/` du projet. Ainsi, si Symfony a été installé depuis l'archive ZIP, le contenu de la nouvelle archive (nouvelle version) peut alors être décompressé dans ce répertoire pour remplacer les fichiers source existants.

Mise à jour de Symfony à l'aide de Subversion

Les utilisateurs de Subversion préféreront certainement mettre à jour leur version de Symfony pour chacun de leur projet en liant cet outil au dernier *tag* Symfony 1.2.

```
$ svn propedit svn:externals lib/vendor/
# symfony http://svn.symfony-project.com/tags/RELEASE_1_2_1/
```

Mettre à jour Symfony est ensuite aussi simple que de changer le tag de la toute dernière version du framework disponible sur le dépôt Subversion du projet. Une autre approche consiste à installer les fichiers source de Symfony comme des ressources externes à l'aide de la propriété `svn:externals` de Subversion. Cette méthode a l'avantage de faire profiter le développeur de la toute dernière version de développement de Symfony à jour, à chaque fois qu'il exécute la commande `svn upragde`.

```
$ svn propedit svn:externals lib/vendor/
# symfony http://svn.symfony-project.com/branches/1.2/
```

Lorsque Symfony est mis à jour, il est recommandé de toujours nettoyer le cache, et tout particulièrement en environnement de production étant donné que de nouvelles classes peuvent être ajoutées au framework.

```
$ php symfony cc
```

Basculer d'une version de Symfony à une autre facilement

Il arrive parfois que les développeurs veuillent tester une nouvelle version de Symfony sans avoir à remplacer celle qui existe déjà pour le projet ; l'objectif étant bien sûr de pouvoir revenir en arrière facilement si cela ne convient pas.

Pour y parvenir, il suffit simplement de télécharger les fichiers source de Symfony dans un autre répertoire du projet (`lib/vendor/symfony_test/` par exemple), puis de modifier le chemin relatif dans la classe de configuration `ProjectConfiguration`, et enfin de procéder au nettoyage du cache. Finalement, pour revenir en arrière, la tâche est aussi simple : rééditer le chemin relatif dans la classe de configuration du projet, supprimer le répertoire des sources de Symfony, et enfin régénérer le cache.

ASTUCE Nettoyer le cache manuellement sans passer par la ligne de commande

Lorsque l'accès au serveur de production est impossible depuis SSH, la seule solution pour simuler le nettoyage du cache est de se connecter à l'aide d'un client FTP, puis de supprimer manuellement tous les répertoires qui se trouvent dans le répertoire `cache/` du projet.

Personnaliser la configuration de Symfony

Configurer l'accès à la base de données

La plupart du temps, la base de données du serveur de production dispose de droits d'accès différents de celle installée localement. Il est donc nécessaire de mettre à jour ces droits pour l'environnement de production en utilisant notamment la commande `configure:database`. Cette tâche automatique a déjà été utilisée au troisième chapitre pour initialiser les codes d'accès aux deux bases de données de Jobeet.

```
$ php symfony configure:database
  ➥ "mysql:host=localhost;dbname=prod_dbname" prod_user prod_pass
```

Cette tâche se contente simplement d'écrire les nouveaux identifiants de connexion à la base de données de production directement dans le fichier de configuration `databases.yml`. L'utilisation de l'interface en ligne de commande n'est bien sûr pas obligatoire, et le fichier `databases.yml` peut aussi être édité manuellement dans un éditeur de texte.

Générer les liens symboliques pour les ressources web

Comme l'application Jobeet utilise des plug-ins qui embarquent des ressources web (*assets*), certains liens symboliques doivent être recréés afin de rendre ces fichiers disponibles depuis un navigateur web. Heureusement, la tâche `plugin:publish-assets` est faite spécialement pour ça car elle permet de régénérer ces liens symboliques, ou de les créer si les plug-ins n'ont pas été installés à partir de la commande `plugin:install`.

```
$ php symfony plugin:publish-assets
```

Personnaliser les pages d'erreur par défaut

Remplacer les pages d'erreur interne par défaut

Avant de basculer les fichiers sur le serveur de production, il est préférable de personnaliser certaines pages par défaut de Symfony comme la page d'erreur 404 « Page non trouvée » ou encore la page d'erreur 500 lorsqu'une erreur interne se produit notamment par le biais des exceptions.

Les pages d'erreur pour le format YAML ont déjà été configurées au chapitre 15 sur les services web en créant les nouveaux fichiers `error.yaml.php` et `exception.yaml.php` dans le répertoire `config/error/`. Le fichier `error.yaml.php` est utilisé par Symfony en environnement de

production tandis que le fichier `exception.yaml.php` sert pour l'environnement de développement. Les pages d'erreur pour le format HTML sont personnalisables en procédant de la même manière, c'est-à-dire en créant deux fichiers `config/error/error.html.php` et `config/error/exception.html.php`.

Personnaliser les pages d'erreur 404 par défaut

La page d'erreur 404 (« Page non trouvée ») peut également être personnalisée en changeant les valeurs des directives de configuration `error_404_action` et `error_404_module` du fichier `settings.yml` de l'application.

Modification des pages d'erreur 404 par défaut de Symfony dans le fichier apps/frontend/config/settings.yml

```
all:
  .actions:
    error_404_module: default
    error_404_action: error404
```

Personnaliser la structure de fichiers par défaut

Nombreux sont les prestataires d'hébergement qui imposent des restrictions sur leurs plates-formes d'hébergement, en particulier sur les serveurs mutualisés pour des raisons de sécurité. Ces restrictions impactent généralement la configuration de PHP ainsi que le nom du répertoire qui représente la racine web, et qui ne peut être modifié.

Modifier le répertoire par défaut de la racine web

Il arrive par convention que ce répertoire soit nommé `public_html/` ou `www/` au lieu de `web/`, ce qui rend impossible le fonctionnement de Symfony. Heureusement, la classe `ProjectConfiguration` du projet possède les méthodes adéquates pour redéfinir les chemins de certains répertoires comme celui de la racine web.

Modification du chemin vers la racine web dans le fichier config/ProjectConfiguration.class.php

```
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    $this->setWebDir($this->getRootDir().'/public_html');
  }
}
```

La méthode `setWebDir()` accepte en paramètre le chemin absolu vers le répertoire qui sert de racine web. Si ce répertoire est déplacé ailleurs, il ne faut pas oublier d'éditer les scripts contrôleurs pour vérifier que les chemins vers le fichier `config/ProjectConfiguration.class.php` sont toujours valides.

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

Modifier les répertoires du cache et des logs

Le framework Symfony n'écrit que dans deux répertoires : `cache/` et `log/`. Pour des raisons de sécurité évidentes, certains hébergeurs (les gratuits entre autres) n'activent pas les permissions d'écriture dans le répertoire principal mais parfois dans un autre qui se trouve en dehors. Si c'est le cas, alors ces deux répertoires peuvent être déplacés en toute sécurité dans un endroit accessible en écriture sur le système de fichiers. Il suffit alors simplement d'indiquer à Symfony où ils se situent sur ce dernier.

Modification des chemins vers les répertoires de cache et de log de Symfony dans le fichier config/ProjectConfiguration.class.php

```
class ProjectConfiguration extends sfProjectConfiguration
{
  public function setup()
  {
    $this->setCacheDir('/tmp/symfony_cache');
    $this->setLogDir('/tmp/symfony_logs');
  }
}
```

De la même manière que la méthode `setWebDir()`, les méthodes `setCacheDir()` et `setLogDir()` prennent en paramètre un chemin absolu vers les répertoires respectifs `cache/` et `log/`.

À la découverte des factories

L'ensemble des pages de cet ouvrage a introduit petit à petit les différents objets du cœur de Symfony comme `sfUser`, `sfRequest`, `sfResponse`, `sfI18N`, `sfRouting` et bien d'autres encore. Ces objets sont automatiquement créés, configurés et gérés par le framework Symfony. Ils sont de plus toujours accessibles depuis l'objet `sfContext`, et comme pour beaucoup de choses dans le framework, ils sont configurables par le biais d'un fichier de configuration encore non étudié jusqu'à maintenant : `factories.yml`. Ce fichier de configuration est lui aussi paramétrable par environnement.

Initialisation des objets du noyau grâce à factories.yml

Le fichier `factories.yml` décrit par environnement la configuration des différents objets du noyau qui doivent être créés automatiquement par l'objet `sfContext`. En effet, lorsque l'objet `sfContext` initialise ses factories, il lit le fichier `factories.yml` afin de connaître quelles classes (`class`) il doit instancier et quels paramètres (`param`) il doit passer au constructeur de chacune d'entre elles.

Extrait de description de l'objet sfResponse dans le fichier de configuration apps/frontend/config/factories.yml

```
response:
  class: sfWebResponse
  param:
    send_http_headers: false
```

Dans cet extrait de code YAML, pour créer l'objet de la réponse, Symfony instancie la classe `sfWebResponse` et passe à son constructeur la valeur de l'option `send_http_header` en guise de paramètre. Être capable de personnaliser soi-même les factories signifie aussi que Symfony donne au développeur la capacité de changer les objets utilisés par défaut par le cœur de Symfony. De la même manière, il est possible de modifier le comportement par défaut de ces classes en changeant les paramètres qui leur sont envoyés.

Les sections suivantes présentent quelques exemples typiques de personnalisation de ces objets à l'initialisation de Symfony, et une annexe entière est consacrée en fin d'ouvrage à la description des différentes directives de configuration sur lesquelles il est possible d'agir.

Modification du nom du cookie de session

Pour reconnaître et manipuler la session courante de l'utilisateur entre chaque page, Symfony utilise un cookie. Ce cookie possède un nom par défaut, qui peut bien évidemment être changé à l'aide du fichier `factories.yml`. Il suffit pour cela d'ajouter la configuration suivante au fichier sous la section `all` pour changer le nom du cookie de session de Jobeet.

Modification du nom du cookie de session dans le fichier apps/frontend/config/factories.yml

```
storage:
  class: sfSessionStorage
  param:
    session_name: jobeet
```

Remplacer le moteur de stockage des sessions par une base de données

Dans Symfony, la classe par défaut pour gérer la session de l'utilisateur courant est `sfSessionStorage`, qui est en réalité une API orientée objet surchargeant le mécanisme natif des sessions de PHP. Par conséquent, les informations de session des utilisateurs sont conservées sur le système de fichiers local. Or, il arrive très souvent qu'il faille gérer les sessions des utilisateurs différemment, en ayant par exemple recours à une base de données dans laquelle tout est centralisé. Grâce aux factories et à la classe `sfPDOSessionStorage`, le mécanisme de gestion des sessions utilisateurs est alors automatiquement déporté de manière totalement transparente vers une table de la base de données.

Configuration d'une base de données pour le stockage des sessions dans le fichier 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
```

La classe `sfPDOSessionStorage` accepte de nouveaux paramètres dans son constructeur qui définissent respectivement :

- le nom de la table dans laquelle sont stockées les sessions ;
- le nom de la base de données qui héberge cette table ;
- le nom de la colonne qui sert de clé primaire dans la table ;
- le nom de la colonne de la table qui accueille les données de session sérialisées ;
- le nom de la colonne qui contient la date de dernière mise à jour de la session.

Grâce à ces cinq informations, le framework Symfony est capable de déporter la gestion des sessions vers une base de données.

Définir la durée de vie maximale d'une session

Toutes les sessions sont éphémères et possèdent donc une durée de vie dans le temps. La durée de vie d'une session détermine en réalité le temps pendant lequel la session reste active entre deux pages consécutives. Si l'utilisateur passe d'une page à une autre dans un temps inférieur

à celui de la durée de la session, alors sa session est perdurée de ce même laps de temps jusqu'à la page suivante, et ainsi de suite.

En revanche, si le temps passé entre deux pages consécutives excède la valeur maximale, alors toutes les informations sont perdues ; la session est considérée comme expirée et doit être réinitialisée. Avec le fichier `factories.yml`, la valeur de cet intervalle de validité de la session peut être modifiée à l'aide de la directive de configuration `timeout` qui fixe la durée à 1 800 secondes, soit 30 minutes.

Modification de la durée de vie d'une session dans le fichier apps/frontend/config/factories.yml

```
user:
  class: myUser
  param:
    timeout: 1800
```

Définir les objets d'enregistrement d'erreur

Par défaut, l'enregistrement des erreurs en environnement de production est désactivé en raison du nom de la classe du logger, `sfNoLogger`, comme le montre la configuration ci-dessous.

Définition des objets de log d'erreur en environnement de production dans le fichier apps/frontend/config/factories.yml

```
prod:
  logger:
    class:   sfNoLogger
    param:
      level:   err
      loggers: ~
```

Néanmoins, l'enregistrement des erreurs dans les fichiers de log du système de fichiers peut être activé en modifiant le nom de la classe par `sfFileLogger` par exemple.

Activation des logs en environnement de production dans le fichier apps/frontend/config/factories.yml

```
logger:
  class: sfFileLogger
  param:
    level: error
    file: %SF_LOG_DIR%/%SF_APP%_%SF_ENVIRONMENT%.log
```

ASTUCE Utiliser des chemins absolus dans un fichier YAML

Dans le fichier de configuration `factories.yml`, les chaînes `%XXX%` sont remplacées par leur valeur équivalente dans l'objet global `sfConfig`. Par exemple, `%SF_APP%` dans un fichier de configuration YAML est similaire à l'appel à `sfConfig::get('sf_app')` dans du code PHP. Cette notation peut aussi être utilisée dans le fichier de configuration `app.yml`. C'est particulièrement utile lorsque l'on a besoin de référencer un chemin dans un fichier de configuration sans avoir à coder en dur ce chemin (`SF_ROOT_DIR`, `SF_WEB_DIR`, etc.).

Déployer le projet sur le serveur de production

Que faut-il déployer en production ?

Le déploiement sur le serveur de production est la toute dernière étape avant que le projet ne démarre véritablement son cycle de vie sur Internet. Cette procédure est capitale car il est important de faire attention à ne pas déployer de fichiers inutiles ou écraser des données existantes envoyées par les utilisateurs comme les logos des sociétés.

Dans un projet Symfony, trois répertoires sont à exclure au moment du transfert des fichiers sur le serveur de production : `cache/`, `log/` et `web/uploads/`. Tous les autres fichiers peuvent être transférés tels quels. Néanmoins, pour des raisons de sécurité évidentes, il convient de ne pas transférer les contrôleurs frontaux non destinés à l'environnement de production comme les fichiers `frontend_dev.php`, `backend_dev.php` et `frontend_cache.php`.

Mettre en place des stratégies de déploiement

Cette section explique comment automatiser et faciliter le déploiement d'une application vers un serveur de production. Par conséquent, il est absolument nécessaire d'avoir les pouvoirs requis sur le serveur de production afin d'y accéder par l'intermédiaire d'une connexion SSH. Si en revanche l'accès au serveur est restreint à un compte FTP, la seule solution possible de déploiement consistera à transférer manuellement tous les fichiers à chaque fois qu'un nouveau processus de déploiement entrera en jeu.

Déploiement à l'aide d'une connexion SSH et rsync

Avec Symfony, la manière la plus simple de déployer une application web est d'utiliser la commande intégrée `project:deploy` qui fait appel aux clients SSH et rsync pour se connecter au serveur et transférer les fichiers d'un ordinateur à un autre. Cependant, l'utilisation de cette tâche automatique requiert la configuration des paramètres de connexion au serveur de production à l'intérieur du fichier `config/properties.ini`. Ce fichier de configuration est capable de définir plusieurs serveurs représentés par les sections entre crochets. Dans le cadre de Jobeet, seul un serveur de production est nécessaire.

Paramétrage des identifiants de connexion au serveur de production dans le fichier config/properties.ini

```
[production]
  host=www.jobeet.org
  port=22
  user=jobeet
  dir=/var/www/jobeet/
  type=rsync
  pass=
```

Une fois les paramètres du serveur établis, l'application peut être déployée à l'aide de la commande `project:deploy`.

```
$ php symfony project:deploy production
```

En réalité la commande précédente ne fait que simuler le transfert. Pour lancer véritablement le déploiement des fichiers sur le serveur de production, l'option `--go` doit être ajoutée explicitement.

```
$ php symfony project:deploy production --go
```

Configurer rsync pour exclure certains fichiers du déploiement

Par défaut, Symfony ne transfèrera ni les répertoires ni les contrôleurs frontaux sensibles de développement mentionnés dans la section précédente. C'est en effet parce que la tâche `project:deploy` exclut automatiquement du processus de déploiement les fichiers et dossiers configurés dans le fichier `config/rsync_exclude.txt`.

Configuration des fichiers et répertoires à exclure du déploiement dans le fichier config/rsync_exclude.txt

```
.svn
/web/uploads/*
/cache/*
/log/*
/web/*_dev.php
```

L'application Jobeet dispose du contrôleur frontal `frontend_cache.php` qui doit lui aussi être exclu lors de la phase de déploiement sur le serveur de production.

ASTUCE **Prérequis avant l'utilisation de la tâche project:deploy**

Avant d'exécuter pour la première fois la commande `project:deploy`, il est nécessaire de se connecter au serveur manuellement afin d'ajouter la clé dans le fichier des serveurs reconnus.

INFORMATION **Configuration du serveur SSH**

Bien que le fichier `properties.ini` accepte la définition d'un mot de passe SSH, il est recommandé de configurer le serveur avec une clé SSH afin de permettre les connexions sans mot de passe.

Exclusion du fichier fontrend_cache.php du processus de déploiement dans le fichier config/rsync_exclude.txt

```
.svn
/web/uploads/*
/cache/*
/log/*
/web/*_dev.php
/web/frontend_cache.php
```

Bien que la tâche `project:deploy` de Symfony soit particulièrement utile et flexible, il arrive parfois qu'elle doive être personnalisée davantage dans le cas d'un déploiement plus complexe et différent selon la configuration et la typologie du serveur. C'est pour cette raison qu'il ne faut surtout pas hésiter à améliorer cette tâche en en créant une nouvelle qui hérite de la classe par défaut.

> ASTUCE **Forcer le déploiement de certains fichiers**
>
> Il est également possible de créer un fichier `config/rsync_include.txt` qui contient la liste des fichiers et répertoires qui doivent obligatoirement être transférés lors du processus de déploiement sur le serveur de production final.

Nettoyer le cache de configuration du serveur de production

Par ailleurs, à chaque fois qu'un site Internet est définitivement déployé sur le serveur de production, il est important de se rappeler de nettoyer au moins le cache de la configuration sur le serveur de production.

```
$ php symfony cc --type=config
```

Si par exemple la configuration de quelques routes a changé, le cache spécifique du routage doit lui aussi être régénéré.

```
$ php symfony cc --type=routing
```

> ASTUCE **Avantage du nettoyage sélectif du cache**
>
> Nettoyer le cache sélectivement permet de conserver certaines parties du cache comme celle des templates par exemple.

En résumé…

Le déploiement d'un projet est la toute dernière étape du cycle de vie d'un développement Symfony, mais cela ne signifie pas nécessairement que c'est la fin. C'est en fait généralement le contraire… Par conséquent, il arrivera très certainement qu'il y ait des nouveaux bogues à corriger ainsi que de nouvelles fonctionnalités à implémenter au même moment. Toutefois, grâce à la structure de Symfony et à ses nombreux outils mis à la disposition des développeurs, la mise à jour d'un site Internet deviendra simple, rapide et sans risque.

Cette conclusion achève la réalisation de l'étude de cas Jobeet et met un terme à cet ouvrage. Après une série de vingt et un chapitres consécutifs révélant pas à pas toutes les fonctionnalités essentielles de Symfony, vous devriez être en mesure de piloter et de développer vos propres projets personnels et professionnels avec le framework. Cet ouvrage constituera par la même occasion une ressource documentaire de référence supplémentaire à toutes celles qui existent aujourd'hui sur le site officiel du projet à l'adresse http://www.symfony-project.org.

Le format YAML

A

MOTS-CLÉS :

- Types de données
- Tableaux et collections
- Configuration dans Symfony

La plupart des fichiers dans Symfony sont écrits en format YAML. D'après le site officiel de YAML, il s'agit d'un « format standard de sérialisation de données, lisible pour tout être humain, pour tous les langages de programmation… ». La syntaxe de YAML est particulièrement riche et ces quelques pages dévoilent la plupart des concepts du format YAML et ce qu'ils permettent de décrire avec les outils de Symfony.

Bien que le format YAML puisse décrire des structures de données imbriquées complexes, cette annexe décrit seulement le jeu de fonctionnalités nécessaires pour utiliser YAML en guise de simple format de fichier de configuration pour Symfony.

YAML est un langage simple dédié à la description de données. Comme PHP, il dispose d'une syntaxe pour les types de données primitifs tels que les chaînes de caractères, les booléens, les nombres décimaux ou encore les nombres entiers. Néanmoins, contrairement à PHP, il est capable de faire la différence entre les tableaux (séquences) et les tables de hachage (mapping).

Les données scalaires

Les sections suivantes décrivent la syntaxe des données scalaires qui sont finalement très proches de la syntaxe de PHP. Ces données incluent entre autres les entiers, les nombres décimaux, les booléens ou les chaînes de caractères.

Les chaînes de caractères

Tous les bouts de code présentés dans cette section décrivent la manière de déclarer des chaînes de caractères en format YAML.

```
A string in YAML
'A singled-quoted string in YAML'
```

Une chaîne de caractères simple peut contenir des apostrophes. Le format YAML impose de doubler les apostrophes intermédiaires pour les échapper.

```
'A single quote '' in a single-quoted string'
```

La syntaxe avec des guillemets est notamment utile lorsque la chaîne de caractères démarre ou se termine par des caractères d'espacement spéciaux tels que les sauts de ligne.

```
"A double-quoted string in YAML\n"
```

Le style avec les guillemets fournit un moyen d'exprimer des chaînes arbitraires en utilisant les séquences d'échappement `\`. C'est notamment pratique lorsqu'il est besoin d'embarquer un `\n` ou un caractères Unicode dans une chaîne.

Lorsqu'une chaîne de caractères contient des retours à la ligne, il est possible d'utiliser le style littéral, marqué par un caractère *pipe*|, qui indique que la chaîne contiendra plusieurs lignes. En mode littéral, les nouvelles lignes sont préservées.

```
| \/ /| |\/| |
  / / | |  | |__
```

Alternativement, les chaînes de caractères peuvent être écrites avec la syntaxe repliée, marquée par un caractère >, où chaque retour à la ligne est remplacé par une espace.

```
>
   This is a very long sentence
   that spans several lines in the YAML
   but which will be rendered as a string
   without carriage returns.
```

Il est bon de remarquer les deux espaces avant chaque ligne dans les exemples précédents. Ils n'apparaîtront pas dans les chaînes de caractères PHP finales.

Les nombres

Cette section décrit les différentes manières de déclarer des valeurs numéraires en format YAML tels que les entiers, les nombres à virgules flottantes, les nombres octaux ou bien l'infini.

Les entiers

Un entier se déclare simplement en indiquant sa valeur.

```
# Un entier
12
```

Les nombres octaux

Un nombre octal se déclare de la même manière qu'un nombre entier en le préfixant par un zéro.

```
# Un nombre octal
014
```

Les nombres hexadécimaux

Un nombre hexadécimal se déclare en préfixant sa valeur par `0x`.

```
# Un nombre hexadécimal
0xC
```

Les nombres décimaux

Un nombre décimal se déclare de la même manière qu'un entier en indiquant sa partie décimale avec un point.

```
# Un nombre décimal
13.4
```

Les nombres exponentiels

Un nombre exponentiel peut aussi être représenté en format YAML de la manière suivante.

```
# Un nombre exponentiel
1.2e+34
```

Les nombres infinis

L'infini se représente à l'aide de la valeur `.inf`.

```
# L'infini
.inf
```

Les valeurs nulles : les NULL

Une valeur nulle se matérialise en YAML avec la valeur `null` ou le caractère tilde `~`.

Les valeurs booléennes

Les valeurs booléennes sont décrites à l'aide des chaînes de caractères `true` et `false`. Il faut également savoir que l'analyseur syntaxique YAML du framework Symfony reconnaît une valeur booléenne si elle est exprimée avec l'une de ces valeurs : `on`, `off`, `yes` et `no`. Néanmoins, il est fortement déconseillé de les utiliser depuis qu'elles ont été supprimées des spécifications de Symfony 1.2.

Les dates

Le format YAML utilise le standard ISO 8 601 pour exprimer les dates.

```
# Une date complète
2001-12-14t21:59:43.10-05:00

# Une date simple
2002-12-14
```

Les collections

Un fichier YAML est rarement utilisé pour décrire uniquement des simples valeurs scalaires. La plupart du temps, c'est pour décrire une collection de valeurs. Une collection peut être exprimée à l'aide d'une liste de valeurs ou avec une association d'éléments. Les séquences et les associations sont toutes deux converties sous forme de tableaux PHP.

Les séquences d'éléments

Les séquences d'éléments s'expriment à l'aide d'un tiret - suivi d'un espace pour chacun de leurs éléments. Le code ci-dessous présente une séquence simple de valeurs.

```
- PHP
- Perl
- Python
```

Après analyse, ce code YAML est converti sous la forme d'un tableau PHP simple identique à celui ci-dessous.

```
array('PHP', 'Perl', 'Python');
```

Les associations d'éléments

Les associations simples

Les associations se représentent à l'aide d'un caractère : suivi d'une espace pour exprimer chaque couple « clé => valeur ».

```
PHP: 5.2
MySQL: 5.1
Apache: 2.2.20
```

Après analyse, ce code YAML est converti sous la forme d'un tableau PHP associatif identique à celui ci-dessous.

```
array('PHP' => 5.2, 'MySQL' => 5.1, 'Apache' => '2.2.20');
```

Il est bon de savoir que chaque clé d'une association peut être exprimée à l'aide de n'importe quelle valeur scalaire valide. D'autre part, le nombre d'espaces après les deux points est complètement arbitraire. Par conséquent, le code ci-dessous est strictement équivalent au précédent.

```
PHP:    5.2
MySQL: 5.1
Apache: 2.2.20
```

Les associations complexes imbriquées

Le format YAML utilise des indentations avec un ou plusieurs espaces pour exprimer des collections de valeurs imbriquées comme le montre le code ci-après.

```
"symfony 1.0":
  PHP:    5.0
  Propel: 1.2
"symfony 1.2":
  PHP:    5.2
  Propel: 1.3
```

Après analyse, ce code YAML est converti sous la forme d'un tableau PHP associatif à deux niveaux identique à celui ci-dessous.

```
array(
  'symfony 1.0' => array(
    'PHP'    => 5.0,
    'Propel' => 1.2,
  ),
  'symfony 1.2' => array(
    'PHP'    => 5.2,
    'Propel' => 1.3,
  ),
);
```

Il y a tout de même une chose importante à retenir lorsque l'on utilise les indentations dans un fichier YAML : les indentations sont toujours réalisées à l'aide d'un ou plusieurs espaces mais jamais avec des tabulations.

Combinaison de séquences et d'associations

Les séquences et les associations peuvent également être combinées de la manière suivante :

```
'Chapter 1':
  - Introduction
  - Event Types
'Chapter 2':
  - Introduction
  - Helpers
```

Syntaxe alternative pour les séquences et associations

Les séquences et les associations possèdent toutes deux une syntaxe alternative sur une ligne à l'aide de délimiteurs particuliers. Ces syntaxes permettent ainsi de se passer des indentations pour marquer les différents *scopes*.

Cas des séquences

Une séquence de valeurs peut aussi s'exprimer à l'aide d'une liste entourée par des crochets `[]` d'éléments séparés par des virgules.

```
[PHP, Perl, Python]
```

Cas des associations

Une association de valeurs peut aussi s'exprimer à l'aide d'une liste entourée d'accolades `{}` de couples clé/valeur séparés par des virgules.

```
{ PHP: 5.2, MySQL: 5.1, Apache: 2.2.20 }
```

Cas des combinaisons de séquences et d'associations

De même que précédemment, la combinaison des séquences et des associations reste possible avec leur syntaxe alternative respective. Par conséquent, il ne faut pas hésiter à les utiliser pour arriver à une meilleure lisibilité du code.

```
'Chapter 1': [Introduction, Event Types]
'Chapter 2': [Introduction, Helpers]

"symfony 1.0": { PHP: 5.0, Propel: 1.2 }
"symfony 1.2": { PHP: 5.2, Propel: 1.3 }
```

Les commentaires

Le format YAML accepte l'intégration de commentaires pour documenter le code en les préfixant par le caractère dièse #.

```
# Commentaire sur une ligne
"symfony 1.0": { PHP: 5.0, Propel: 1.2 } # Commentaire à la fin
d'une ligne
"symfony 1.2": { PHP: 5.2, Propel: 1.3 }
```

Les commentaires sont simplement ignorés par l'analyseur syntaxique YAML et n'ont pas besoin d'être indentés selon le niveau d'imbrications courant dans une collection.

Les fichiers YAML dynamiques

Dans Symfony, un fichier YAML peut contenir du code PHP qui sera ensuite évalué juste avant que l'analyse du code YAML finale ne se produise.

```
1.0:
  version: <?php echo file_get_contents('1.0/VERSION')."\n" ?>
1.1:
  version: "<?php echo file_get_contents('1.1/VERSION') ?>"
```

Il faut bien garder à l'esprit ces quelques petites astuces lorsque du code PHP est ajouté au fichier YAML afin ne pas risquer de désordonner l'indentation.

- L'instruction `<?php ?>` doit toujours démarrer une ligne ou être embarquée dans une valeur.
- Si une instruction `<?php ?>` termine une ligne, alors il est nécessaire de générer explicitement une nouvelle ligne en sortie à l'aide du marqueur `\n`.

Exemple complet récapitulatif

L'exemple ci-dessous illustre la plupart des notations YAML expliquées tout au long de cette annexe sur le format YAML.

```
"symfony 1.0":
  end_of_maintainance: 2010-01-01
  is_stable:           true
  release_manager:     "Grégoire Hubert"
  description: >
    This stable version is the right choice for projects
    that need to be maintained for a long period of time.
  latest_beta:         ~
    latest_minor:        1.0.20
    supported_orms:      [Propel]
    archives:            { source: [zip, tgz], sandbox: [zip,
tgz] }

"symfony 1.2":
  end_of_maintainance: 2008-11-01
  is_stable:           true
  release_manager:     'Fabian Lange'
  description: >
    This stable version is the right choice
    if you start a new project today.
   latest_beta:         null
   latest_minor:        1.2.5
   supported_orms:
     - Propel
     - Doctrine
   archives:
     source:
       - zip
       - tgz
     sandbox:
       - zip
       - tgz
```

Le fichier de configuration settings.yml

B

MOTS-CLÉS :

- Configuration de Symfony
- Format YAML
- Fichier settings.yml

La plupart des aspects de Symfony peuvent être configurés à travers un fichier de configuration écrit en YAML ou avec du code PHP pur.

Cette annexe est consacrée à la description de tous les paramètres de configuration d'une application qui se trouvent dans le fichier de configuration principal `settings.yml` du répertoire `apps/APPLICATION/config/`.

Comme il l'a été mentionné dans l'introduction, le fichier de configuration `settings.yml` d'une application est paramétrable par environnement, et bénéficie du principe de configuration en cascade. Chaque environnement dispose de deux sous-sections : `.actions` et `.settings`. Toutes les directives de configuration se situent principalement dans la sous-section `.settings` à l'exception des actions par défaut qui sont rendues pour certaines pages communes.

Les paramètres de configuration du fichier settings.yml

Configuration de la section .actions

La sous-section `.actions` du fichier de configuration `settings.yml` contient quatre directives de configuration listées ci-dessous. Chacune d'entre elles sera décrite indépendamment dans la suite de cette annexe.

- `error_404`
- `login`
- `secure`
- `module_disabled`

Configuration de la section .settings

La sous-section `.settings` du fichier de configuration `settings.yml` contient vingt-deux directives de configuration listées ci-dessous. Chacune d'entre elles sera décrite indépendamment dans la suite de cette annexe.

- `cache`
- `charset`
- `check_lock`
- `check_symfony_version`
- `compressed`
- `csrf_secret`
- `default_culture`
- `default_timezone`
- `enabled_modules`
- `error_reporting`
- `escaping_strategy`

- escaping_method
- etag
- i18n
- logging_enabled
- no_script_name
- max_forwards
- standard_helpers
- strip_comments
- use_database
- web_debug
- web_debug_web_dir

La sous-section .actions

Configuration par défaut

Le code ci-dessous donne le détail de la configuration par défaut définie par Symfony pour les directives de cette sous-section.

```
default:
  .actions:
    error_404_module:       default
    error_404_action:       error404

    login_module:           default
    login_action:           login

    secure_module:          default
    secure_action:          secure

    module_disabled_module: default
    module_disabled_action: disabled
```

La sous-section `.actions` définit les actions à exécuter lorsque des pages communes doivent être rendues. Chaque définition se décompose en deux directives de configuration. La première indique le module concerné (elle est suffixée par `_module`), tandis que la seconde indique l'action à exécuter dans ce module (elle est suffixée par `_action`). Les parties qui suivent décrivent l'une après l'autre ces directives de configuration de l'application.

error_404

L'action `error_404` est exécutée à chaque fois qu'une page d'erreur 404 doit être rendue.

login

L'action `login` est exécutée lorsque l'utilisateur tente d'accéder à une page alors qu'il n'est pas authentifié. Généralement, la page affichée par cette directive est celle qui contient un formulaire d'identification.

secure

L'action `secure` est exécutée lorsque l'utilisateur tente d'accéder à une page pour laquelle il n'a pas les droits d'accès nécessaires.

module_disabled

L'action `module_disabled` est exécutée lorsque l'utilisateur demande une page d'un module désactivé pour l'application.

La sous-section .settings

La sous-section `.settings` est l'endroit où toute la configuration du framework est définie. Les parties qui suivent décrivent tous les paramètres possibles qui sont, à cette occasion, grossièrement triés par ordre d'importance. Toutes les valeurs des paramètres de cette sous-section son disponibles depuis n'importe quel endroit du code par le biais de l'objet `sfConfig`, et sont préfixées par `sf_`. Par exemple, pour connaître la valeur de la directive `charset`, il suffit de l'appeler de cette manière :

```
sfConfig::get('sf_charset');
```

escaping_strategy

La valeur par défaut de la directive `escaping_strategy` est `off`.

La directive de configuration `escaping_strategy` est un booléen qui détermine si le sous-framework d'échappement des valeurs de sortie doit être activé ou non. Quand il est activé, toutes les variables rendues disponibles dans les templates sont automatiquement échappées par l'appel au helper défini par le paramètre `escaping_method` décrit plus bas.

La directive `escaping_method` définit le helper par défaut utilisé par Symfony, mais celui-ci peut bien sûr être redéfini au cas par cas lorsqu'il s'agit de rendre une variable dans une balise JavaScript par exemple. Le sous-framework d'échappement utilise la valeur du paramètre `charset` pour l'échappement.

Il est fortement recommandé de changer la valeur par défaut à la valeur `on`.

ASTUCE **Activer automatiquement la stratégie d'échappement**

La valeur de la stratégie d'échappement des données peut être définie automatiquement au moment de la création de l'application en ajoutant l'option `--escaping_strategy` à la commande `generate:app`.

escaping_method

La valeur par défaut de la directive `escaping_method` est `ESC_SPECIALCHARS`.

La directive de configuration `escaping_method` définit la fonction par défaut à utiliser pour automatiser l'échappement des variables dans les templates (voir la directive `escaping_strategy` plus haut). Ce paramètre accepte l'une des valeurs suivantes : `ESC_SPECIALCHARS`, `ESC_RAW`, `ESC_ENTITIES`, `ESC_JS`, `ESC_JS_NO_ENTITIES` et `ESC_SPECIALCHARS` ; sinon il faut créer une fonction spécifique pour l'échappement.

La plupart du temps, la valeur par défaut suffit. Le helper `ESC_ENTITIES` peut aussi être utilisé, particulièrement lorsqu'il s'agit de travailler avec des langues anglaises ou européennes.

csrf_secret

La valeur par défaut de la directive `csrf_secret` est `false`.

La directive de configuration `csrf_secret` permet de définir un jeton unique pour l'application. Lorsqu'elle n'est pas à la valeur false, elle active la protection contre les vulnérabilités CSRF dans tous les formulaires générés à partir du framework de formulaire de Symfony. Ce paramètre est également embarqué dans le helper `link_to()` lorsqu'il a besoin de convertir un lien en formulaire afin de simuler la méthode HTTP `PUT` par exemple.

Il est fortement recommandé de changer la valeur par défaut par un jeton unique.

ASTUCE **Activer automatiquement la protection CSRF**

La valeur du jeton contre les vulnérabilités CSRF peut être définie automatiquement au moment de la création de l'application en ajoutant à la commande `--csrf_secret` l'option : `generate:app`.

charset

La valeur par défaut de la directive `charset` est `utf-8`.

La directive de configuration `charset` définit la valeur de l'encodage qui sera utilisé par défaut dans tout le framework de la réponse (en-tête `Content-Type`) jusqu'aux informations échappées dans les templates. La plupart du temps, la configuration de ce paramètre suffit.

enabled-modules

La valeur par défaut de la directive `enabled_modules` est `[default]`.

La directive de configuration `enabled_modules` est un tableau des noms des modules à activer pour l'application courante. Les modules définis dans les plug-ins ou dans le cœur de Symfony ne sont pas activés par défaut, et doivent absolument être listés dans ce paramètre pour être accessibles.

Ajouter un nouveau module est aussi simple que de l'enregistrer dans la liste. L'ordre des modules activés n'a aucune incidence.

```
enabled_modules: [default, sfGuardAuth]
```

Le module `default` défini par défaut dans le framework contient toutes les actions par défaut qui sont déclarées dans la sous-section `.actions` du fichier de configuration `settings.yml`. Il est vivement recommandé de personnaliser chacune de ces actions, puis de supprimer le module `default` de la liste de ce paramètre.

REMARQUE

Définir un fuseau horaire par défaut

Si aucun fuseau horaire n'a été défini pour cette directive de configuration, il est fortement recommandé de le déclarer dans le fichier de configuration `php.ini` du serveur étant donné que Symfony essaiera de déterminer le fuseau horaire approprié d'après la valeur retournée par la fonction PHP `date_default_timezone_get()`.

default_timezone

La valeur par défaut de la directive `default_timezone` est `none`.

La directive de configuration `default_timezone` définit le fuseau horaire utilisé par PHP. La valeur peut être n'importe quel fuseau horaire reconnu par PHP (voir http://www.php.net/manual/en/class.datetimezone.php).

ASTUCE

Paramétrage du cache des pages HTML

La configuration du système de cache des pages HTML doit être réalisée au niveau des sections `view_cache` et `view_cache_manager` du fichier de configuration `factories.yml` (voir annexe 3). La configuration par niveau de granularité la plus fine est gérée quant à elle par le biais du fichier de configuration `cache.yml`.

cache

La valeur par défaut de la directive `cache` est `off`.

La directive de configuration `cache` active ou non le cache des pages HTML.

etag

La valeur par défaut de la directive `etag` est `on`, à l'exception des environnements `dev` et `test` pour lesquels elle est désactivée.

La directive de configuration `etag` active ou désactive la génération automatique des en-têtes HTTP `ETag`. Les `ETag` générés par Symfony sont de simples calculs MD5 du contenu de la réponse.

i18n

La valeur par défaut de la directive `i18n` est `off`.

La directive de configuration `i18n` est un booléen qui active ou non le sous-framework d'internationalisation de Symfony. Pour les applications internationalisées, la valeur de ce paramètre doit être placée à `on`.

ASTUCE **Paramétrage du système d'I18N**

La configuration générale du système d'internationalisation doit être réalisée au niveau de la section `i18n` du fichier `factories.yml` (voir annexe C).

default_culture

La valeur par défaut de la directive `default_culture` est `en`.

La directive de configuration `default_culture` définit la culture par défaut que le sous-framework d'internationalisation utilisera. Ce paramètre accepte n'importe quelle valeur valide de culture.

standard_helpers

La valeur par défaut de la directive `standard_helpers` est `[Partial, Cache, Form]`.

La directive de configuration `standard_helpers` définit tous les groupes de helpers à charger automatiquement pour tous les templates de l'application. Les valeurs indiquées sont les noms de chaque groupe de helpers sans leur suffixe `Helper`.

no_script_name

La valeur par défaut de la directive `no_script_name` est `on` pour l'environnement de production de la toute première application créée dans le projet, et `off` pour tous les autres.

La directive de configuration `no_script_name` détermine si le nom du contrôleur frontal doit apparaître ou non dans l'URL avant les URLs générées par Symfony. Par défaut, il est fixé à la valeur `on` par la tâche automatique `generate:app` pour l'environnement `prod` de la toute première application créée.

De toute évidence, seulement une application et un environnement peuvent avoir ce paramètre à la valeur `on` si tous les contrôleurs frontaux se trouvent dans le même répertoire (`web/` par défaut). Pour avoir plusieurs applications avec la directive de configuration `no_script_name` à la valeur `on`, il suffit de déplacer le ou les contrôleurs frontaux sous un sous-répertoire du répertoire `web/` racine.

logging_enabled

La valeur par défaut de la directive `logging_enabled` est `on` pour tous les environnements à l'exception de l'environnement de production `prod`.

La directive de configuration `logging_enabled` active le sous-framework d'enregistrement des traces de logs. Fixer la valeur de ce paramètre à `false` permet de détourner complètement le mécanisme d'enregistrement des logs et ainsi d'améliorer légèrement les performances.

ASTUCE **Paramétrage du système d'enregistrement des traces de logs**

La configuration générale du système d'enregistrement des logs doitêtre réalisée dans le fichier de configuration `factories.yml` (voir annexe 3) afin de déterminer son niveau de granularité le plus fin.

web_debug

La valeur par défaut de la directive `web_debug` est `on` pour tous les environnements à l'exception de l'environnement de développement `dev`.

La directive de configuration `web_debug` active la génération de la barre de débogage de Symfony. La barre de débogage est ajoutée à toutes les pages web ayant la valeur HTML dans l'en-tête de type de contenu.

error_reporting

Les valeurs par défaut de la directive `error_reporting` sont les suivantes en fonction de l'environnement d'exécution du script :

- `prod` : `E_PARSE | E_COMPILE_ERROR | E_ERROR | E_CORE_ERROR | E_USER_ERROR`
- `dev` : `E_ALL | E_STRICT`
- `test` : `(E_ALL | E_STRICT) ^ E_NOTICE`
- `default` : `E_PARSE | E_COMPILE_ERROR | E_ERROR | E_CORE_ERROR | E_USER_ERROR`

REMARQUE
En savoir plus sur les niveaux d'erreurs

Le site officiel de PHP donne des informations complémentaires sur comment utiliser les opérateurs bit-à-bit à l'adresse http://www.php.net/language.operators.bitwise.

La directive de configuration `error_reporting` contrôle le niveau d'erreurs à rapporter dans le navigateur et à écrire dans les fichiers de logs. La configuration actuelle est la plus sensible, c'est pourquoi elle ne devrait pas avoir à être modifiée.

REMARQUE **Désactivation des erreurs dans le navigateur**

L'affichage des erreurs dans le navigateur est automatiquement désactivé pour les applications qui ont la directive de configuration `debug` désactivée, ce qui est le cas par défaut pour l'environnement de production.

compressed

La valeur par défaut de la directive `compressed` est `off`.

La directive de configuration `compressed` active ou non la compression automatique du contenu de la réponse. Si la valeur de ce paramètre est à `on`, alors Symfony utilisera la fonction native `ob_gzhandler()` de PHP en guise de fonction de rappel à la fonction `ob_start()`.

Il est néanmoins recommandé de la garder à la valeur `off`, et d'utiliser à la place le mécanisme de compression natif supporté par le navigateur.

use_database

La valeur par défaut de la directive `use_database` est `on`.

La directive de configuration `use_database` indique si oui ou non l'application nécessite l'usage d'une base de données.

check_lock

La valeur par défaut de la directive `check_lock` est `off`.

La directive de configuration `check_lock` active ou non le système de verrouillage de l'application déclenché par certaines tâches automatiques telles que `cache:clear`.

Si la valeur de ce paramètre de configuration est à `on`, alors toutes les requêtes en direction des applications désactivées seront automatiquement redirigées vers la page `lib/exception/data/unavailable.php`.

check_symfony_version

La valeur par défaut de la directive `check_symfony_version` est `off`.

La directive de configuration `check_symfony_version` active ou non le contrôle de la version de Symfony à chaque requête exécutée. Si ce paramètre est activé, Symfony vide le cache automatiquement lorsque le framework est mis à jour.

Il est vivement recommandé de ne pas fixer la valeur de cette directive à `on` étant donné qu'elle ajoute un léger coût supplémentaire sur les performances et qu'il est simple de nettoyer le cache lorsqu'une nouvelle version du projet est déployée. Ce paramètre est seulement utile si plusieurs projets partagent le même code source de Symfony, ce qui n'est véritablement pas recommandé.

web_debug_dir

La valeur par défaut de la directive `web_debug_dir` est `/sf/sf_web_debug`.

La directive de configuration `web_debug_dir` définit le répertoire qui contient toutes les ressources web nécessaires au bon fonctionnement de la barre de débogage de Symfony telles que les feuilles de style CSS, les fichiers JavaScript ou encore les images.

strip_comments

La valeur par défaut de la directive `strip_comments` est `on`.

La directive de configuration `strip_comments` détermine si Symfony devrait supprimer les commentaires lorsqu'il compile les classes du noyau. Ce paramètre est uniquement utilisé si le paramètre de configuration `debug` de l'application est fixé à la valeur `off`.

Si des pages blanches apparaissent uniquement en environnement de production, alors il convient de réessayer en définissant ce paramètre de configuration à la valeur `off`.

max_forwards

La valeur par défaut de la directive `max_forward` est `5`.

La directive de configuration `max_forward` détermine le nombre maximum de redirections internes (`forward()` dans les actions) autorisées avant que Symfony ne lève une exception. Ce paramètre de configuration est une sécurité pour se prémunir des boucles sans fin.

Le fichier de configuration factories.yml

C

MOTS-CLÉS :

- Configuration de Symfony
- Format YAML
- Fichier factories.yml

Tous les objets internes de Symfony nécessaires au bon traitement des requêtes comme `request`, `response` ou `user`, sont générés automatiquement par l'objet `sfContext`. Or, il arrive parfois que les classes utilisées pour construire ces objets ne suffisent pas pour subvenir aux besoins de l'application ; elles doivent alors être remplacées par des classes personnalisées.

Grâce au fichier de configuration `factories.yml` de l'application, Symfony autorise le développeur à définir lui-même son contexte d'exécution.

Introduction à la notion de « factories »

Les *factories* sont les objets du noyau de Symfony et qui sont nécessaires au framework durant tout le cycle de vie du traitement de la requête. Ces objets sont tous définis dans le fichier de configuration `factories.yml` du répertoire `apps/APPLICATION/config/` et sont toujours accessibles depuis n'importe où par l'intermédiaire de l'objet `sfContext`.

```
sfContext::getInstance()->getUser();
```

Le fichier de configuration `factories.yml` d'une application est paramétrable par environnement, et bénéficie du principe de configuration en cascade. Il peut également prendre en compte les constantes globales définies par Symfony.

Lorsque l'objet `sfContext` initialise les factories, il lit le fichier `factories.yml` pour en découvrir les noms des classes (`class`) qu'il doit instancier et la liste des paramètres (`param`) qu'il doit transmettre au constructeur respectif de ces dernières. La description générale en format YAML d'un objet factory est la suivante :

```
FACTORY_NAME:
  class: CLASS_NAME
  param: { ARRAY OF PARAMETERS }
```

REMARQUE Classe de conversion du fichier factories.yml en code PHP

La conversion du fichier de configuration `factories.yml` en PHP est réalisée par la classe PHP `sfFactoryConfigHandler`.

Être capable de personnaliser les factories signifie aussi utiliser des classes personnalisées pour les objets du cœur de Symfony à la place de ceux qui sont créés par défaut. Par conséquent, cette liberté de configuration offre au développeur la possibilité de changer les comportements par défaut de ces objets en leur passant d'autres paramètres.

Présentation du fichier factories.yml

Configuration du service request

L'initialisation de l'objet *requête* par le contexte d'exécution est assurée par quatre paramètres de configuration du fichier `factories.yml`. Chaque paramètre de configuration listé ci-dessous sera décrit indépendamment dans la suite de cette annexe.

- `path_info_array`
- `path_info_key`
- `formats`
- `relative_url_root`

Configuration du service response

L'initialisation de l'objet *réponse* par le contexte d'exécution est assurée par trois paramètres de configuration du fichier `factories.yml`. Chaque paramètre de configuration listé ci-dessous sera décrit indépendamment dans la suite de cette annexe.

- `send_http_headers`
- `charset`
- `http_protocol`

Configuration du service user

L'initialisation de l'objet *utilisateur* par le contexte d'exécution est assurée par trois paramètres de configuration du fichier `factories.yml`. Chaque paramètre de configuration listé ci-dessous sera décrit indépendamment dans la suite de cette annexe.

- `timeout`
- `use_flash`
- `default_culture`

Configuration du service storage

L'initialisation de l'objet de *session* par le contexte d'exécution est assurée par treize paramètres de configuration du fichier `factories.yml`. Chaque paramètre de configuration listé ci-dessous sera décrit indépendamment dans la suite de cette annexe.

- `auto_start`
- `session_name`
- `session_cache_limiter`
- `session_cookie_lifetime`
- `session_cookie_path`
- `session_cookie_domain`
- `session_cookie_secure`
- `session_cookie_httponly`
- `database`
- `db_table`
- `db_id_col`
- `db_data_col`
- `db_time_col`

Configuration du service i18n

L'initialisation de l'objet d'*internationalisation* par le contexte d'exécution est assurée par cinq paramètres de configuration du fichier `factories.yml`. Chaque paramètre de configuration listé ci-dessous sera décrit indépendamment dans la suite de cette annexe.

- `source`
- `debug`
- `untranslated_prefix`
- `untranslated_suffix`
- `cache`

Configuration du service routing

L'initialisation de l'objet de *routage* par le contexte d'exécution est assurée par sept paramètres de configuration du fichier `factories.yml`. Chaque paramètre de configuration listé ci-dessous sera décrit indépendamment dans la suite de cette annexe.

- `variable_prefixes`
- `segment_separators`
- `generate_shortest_url`
- `extra_parameters_as_query_string`
- `cache`
- `suffix`
- `load_configuration`

Configuration du service logger

L'initialisation de l'objet d'*enregistrement des traces de log* par le contexte d'exécution est assurée par deux paramètres de configuration du fichier `factories.yml`. Chaque paramètre de configuration listé ci-dessous sera décrit indépendamment dans la suite de cette annexe.

- `level`
- `loggers`

Le service request

Configuration par défaut

Le service `request` est accessible grâce à l'accesseur `getRequest()` de l'objet `sfContext`.

```
sfContext::getInstance()->getRequest()
```

La configuration par défaut du service `request` est la suivante :

```
request:
  class: sfWebRequest
  param:
    logging:           %SF_LOGGING_ENABLED%
    path_info_array:   SERVER
    path_info_key:     PATH_INFO
    relative_url_root: ~
    formats:
      txt: text/plain
      js:   [application/javascript, application/x-javascript,
text/javascript]
      css: text/css
      json: [application/json, application/x-json]
      xml: [text/xml, application/xml, application/x-xml]
      rdf: application/rdf+xml
      atom: application/atom+xml
```

path_info_array

L'option `path_info_array` définit le tableau PHP superglobal qui doit être utilisé pour retrouver les informations. Sur certaines configurations de serveur web, il arrive parfois que l'on veuille changer la valeur par défaut `SERVER` en `ENV`.

path_info_key

L'option `path_info_key` définit la clé dans le tableau PHP superglobal avec laquelle il est possible de retrouver l'information `PATH_INFO`. Les utilisateurs de serveurs web `IIS` configurés avec un moteur de réécriture d'URLs comme `IIFR` ou `ISAPI` devront certainement changer la valeur de cette directive de configuration par `HTTP_X_REWRITE_URL`.

ASTUCE **Utiliser la méthode setFormat() de l'objet request**

Au lieu de redéfinir cette directive de configuration, il est possible d'avoir recours à la méthode `setFormat()` de la classe de l'objet de la requête.

formats

L'option `formats` définit un tableau associatif d'extensions et leurs valeurs respectives d'en-tête de types de contenu. Ce paramètre de configuration est utilisé par le framework pour gérer automatiquement l'en-tête `Content-Type` de la réponse, en partant de l'extension de l'URL de la requête.

relative_root_url

L'option `relative_root_url` définit la valeur de la partie de l'URL qui se trouve avant le nom du contrôleur frontal. La plupart du temps, cette valeur est déduite automatiquement par le framework, ce qui implique qu'il n'est pas nécessaire de la modifier.

Le service response

Configuration par défaut

Le service `response` est accessible grâce à l'accesseur `getResponse()` de l'objet `sfContext`.

```
sfContext::getInstance()->getResponse()
```

La configuration par défaut du service `response` est la suivante :

```
response:
  class: sfWebResponse
  param:
    logging:           %SF_LOGGING_ENABLED%
    charset:           %SF_CHARSET%
    send_http_headers: true
```

Le code ci-dessous donne la configuration par défaut du service `response` en environnement de test.

```
response:
     class: sfWebResponse
     param:
       send_http_headers: false
```

send_http_headers

L'option `send_http_headers` définit si la réponse a besoin d'envoyer les en-têtes HTTP avec son contenu. Ce paramètre de configuration est essentiellement utilisé en environnement de test étant donné que l'envoi

des en-têtes est réalisé par la fonction native PHP `header()` qui provoque des avertissements lorsque des en-têtes sont envoyés après les premières sorties au navigateur.

charset

L'option `charset` définit l'encodage à utiliser pour la réponse. Par défaut, la valeur de cette directive de configuration est la même que celle définie à l'entée `charset` dans le fichier de configuration `settings.yml` de l'application.

http_protocol

L'option `http_protocol` détermine la version du protocole HTTP à utiliser pour transmettre la réponse. Par défaut, Symfony ira chercher cette valeur dans la variable superglobale `$_SERVER['SERVER_PROTOCOL']` si elle existe, ou utilisera `HTTP/1.0` par défaut.

Le service user

Configuration par défaut

Le service `user` est accessible grâce à l'accesseur `getUser()` de l'objet `sfContext`.

```
sfContext::getInstance()->getUser()
```

La configuration par défaut du service `user` est la suivante :

```
user:
  class: myUser
  param:
    timeout:         1800
    logging:         %SF_LOGGING_ENABLED%
    use_flash:       true
    default_culture: %SF_DEFAULT_CULTURE%
```

Par défaut, la classe `myUser` hérite des propriétés et des méthodes de la classe `sfBasicSecurityUser`, qui peut être configurée dans le fichier de configuration `security.yml`.

timeout

L'option `timeout` définit le temps maximum pendant lequel l'utilisateur dispose de son authentification et de ses droits d'accès. Cette valeur n'est pas relative à celle définie dans la session. Ce paramètre de configuration est seulement utilisé par les classes de gestion de l'utilisateur qui héritent de la classe de base `sfBasicSecurityUser`, ce qui est le cas pour la classe autogénérée `myUser` de chaque application.

use_flash

L'option `use_flash` active ou désactive l'utilisation des messages éphémères stockés dans la session de l'utilisateur entre deux requêtes HTTP.

default_culture

L'option `default_culture` détermine la culture par défaut à assigner à l'utilisateur lorsque celui-ci arrive pour la première fois sur le site Internet. Par défaut, cette valeur est récupérée de la directive de configuration `default_culture` du fichier de configuration `settings.yml`, ce qui est généralement le comportement de la plupart des applications.

REMARQUE **Éviter les effets de bord avec l'objet User**

Pour éviter des effets de bord étranges, la classe de gestion de l'utilisateur force automatiquement la durée de vie maximale du ramasse-miettes (*garbage collector*) de la session (`session.gc_maxlifetime`) à une valeur strictement supérieure à celle du paramètre `timeout`.

Le service storage

Configuration par défaut

Le service `storage` est utilisée par le service `user` pour assurer la persistance des données de session de l'utilisateur entre chaque requête HTTP, et est accessible grâce à l'accesseur `getStorage()` de l'objet `sfContext`.

```
sfContext::getInstance()->getStorage()
```

La configuration par défaut du service `storage` est la suivante :

```
storage:
  class: sfSessionStorage
  param:
    session_name: symfony
```

Une configuration par défaut spéciale pour l'environnement de test est également mise en place. Elle est décrite dans le code ci-dessous.

```
storage:
  class: sfSessionTestStorage
  param:
    session_path: %SF_TEST_CACHE_DIR%/sessions
```

auto_start

L'option `auto_start` active ou désactive le démarrage automatique de la session PHP par le biais de la fonction `session_start()`.

session_name

L'option `session_name` définit le nom du cookie de session utilisé par Symfony pour sauvegarder l'identifiant de session de l'utilisateur. Par défaut, le nom est Symfony, ce qui signifie que toutes les applications du projet partagent le même cookie, et par conséquent les authentifications et droits d'accès correspondants.

Paramètres de la fonction session_set_cookie_params()

Le service `storage` fait appel à la fonction PHP `session_set_cookie_params()` avec les valeurs des options suivantes :

- `session_cookie_lifetime` : durée de vie totale du cookie de session définie en secondes ;
- `session_cookie_path` : domaine de validité du cookie sur le serveur. La valeur `/` indique que le cookie est valable sur tout le domaine ;
- `session_cookie_domain` : domaine du cookie, par exemple, www.php.net. Pour rendre les cookies visibles par tous les sous-domaines, la valeur doit être préfixée d'un point comme `.php.net` ;
- `session_cookie_secure` : si la valeur est `true`, le cookie sera envoyé sur des connexions sécurisées ;
- `session_cookie_httponly` : si la valeur est `true`, PHP attendra d'envoyer le drapeau `httponly` lorsqu'il paramétrera le cookie de session.

session_cache_limiter

Si l'option `session_cache_limiter` est définie, la fonction native de PHP `session_cache_limiter()` sera appelée et la valeur de cette directive de configuration sera passée en argument de la fonction.

REMARQUE Documentation de la fonction session_set_cookie_params()

La description de chaque option de configuration de la fonction `session_set_cookie_params()` provient directement de la documentation officielle en ligne disponible à l'adresse http://fr.php.net/session-set-cookie-params.

Options de stockage des sessions en bases de données

Lorsque l'on utilise un système de stockage des sessions qui hérite de la classe `sfDatabaseSessionStorage`, certaines options spécifiques sont disponibles :

- `database` : le nom de la base de données (obligatoire) ;
- `db_table` : le nom de la table qui stocke les données de session (obligatoire) ;
- `db_id_col` : le nom de la colonne qui contient la clé primaire (`sess_id` par défaut) ;
- `db_data-col` : le nom de la colonne qui contient les données de session sérialisées (`sess_data` par défaut) ;
- `db_time_col` : le nom de la colonne dans laquelle est stockée le date de la session (`sess_time` par défaut).

Le service view_cache_manager

Le service `view_cache_manager` est accessible grâce à l'accesseur `getViewCacheManager()` de l'objet `sfContext`.

```
sfContext::getInstance()->getViewCacheManager()
```

La configuration par défaut du service `view_cache_manager` est la suivante :

```
view_cache_manager:
  class: sfViewCacheManager
```

Cette factory est uniquement initialisée si le paramètre de configuration `cache` est défini à la valeur `on`. La configuration du gestionnaire de cache des pages HTML est réalisée via le service `view_cache` qui définit l'objet cache sous-jacent à utiliser pour le cache des vues.

Le service view_cache

Le service `view_cache` est accessible depuis l'accesseur `getViewCacheManager()` de l'objet `sfContext`. La configuration par défaut du service `view_cache` est la suivante :

```
view_cache:
  class: sfFileCache
  param:
    automatic_cleaning_factor: 0
    cache_dir:                 %SF_TEMPLATE_CACHE_DIR%
    lifetime:                  86400
    prefix:                    %SF_APP_DIR%/template
```

Cette factory est uniquement initialisée si le paramètre de configuration `cache` est défini à la valeur `on`. Le service `view_cache` déclare une classe qui doit absolument hériter de `sfCache`.

Le service i18n

Configuration par défaut

Le service `i18n` est accessible grâce à l'accesseur `getI18N()` de l'objet `sfContext`.

```
sfContext::getInstance()->getI18N()
```

La configuration par défaut du service `i18n` est la suivante :

```
i18n:
  class: sfI18N
  param:
    source:               XLIFF
    debug:                off
    untranslated_prefix: "[T]"
    untranslated_suffix: "[/T]"
    cache:
      class: sfFileCache
      param:
        automatic_cleaning_factor: 0
        cache_dir:                 %SF_I18N_CACHE_DIR%
        lifetime:                  31556926
        prefix:                    %SF_APP_DIR%/i18n
```

Cette factory est uniquement initialisée si le paramètre de configuration `i18n` est défini à la valeur `on`.

source

L'option `source` définit le type de container des traductions au choix parmi `XLIFF`, `MySQL`, `SQLite` ou `gettext`.

debug

L'option `debug` active ou désactive le débogage. Si cette directive de configuration est activée, alors les messages non traduits seront décorés avec un préfixe et un suffixe (voir ci-dessous).

untranslated_prefix

L'option `untranslated_prefix` définit un préfixe à utiliser pour décorer les messages non traduits.

untranslated_sufix

L'option `untranslated_sufix` définit un suffixe à utiliser pour décorer les messages non traduits.

cache

L'option `cache` définit une factory de cache anonyme à utiliser pour mettre en cache les données internationalisées.

Le service routing

Configuration par défaut

Le service `routing` est accessible grâce à l'accesseur `getRouting()` de l'objet `sfContext`.

```
sfContext::getInstance()->getRouting()
```

La configuration par défaut du service `routing` est la suivante :

```
routing:
  class: sfPatternRouting
  param:
    load_configuration:              true
    suffix:                          ''
    default_module:                  default
    default_action:                  index
    debug:                           %SF_DEBUG%
    logging:                         %SF_LOGGING_ENABLED%
    generate_shortest_url:           true
    extra_parameters_as_query_string: true
```

```
cache:
  class: sfFileCache
  param:
    automatic_cleaning_factor: 0
    cache_dir:                 %SF_CONFIG_CACHE_DIR%/routing
    lifetime:                   31556926
    prefix:                     %SF_APP_DIR%/routing
```

variables_prefix

La valeur par défaut de l'option `variables_prefix` est `:`.

L'option `variables_prefix` définit la liste des caractères qui démarre le nom d'une variable dans un motif d'URL d'une route.

segment_separators

La valeur par défaut de l'option `segment_separators` est `/` et `.`.

L'option `segment_separators` définit la liste des séparateurs de segments des routes. La plupart du temps, cette option n'est pas redéfinie pour tout le framework de routage mais uniquement pour une seule route particulière.

generate_shortest_url

La valeur par défaut de l'option `generate_shortest_url` est `true` pour les nouveaux projets et `false` pour les projets mis à jour.

Si la valeur de l'option `generate_shortest_url` est fixée à la valeur `true`, le sous-framework de routage génèrera la route la plus courte possible. Cette directive de configuration doit en revanche rester à la valeur `false` afin de conserver une compatibilité rétrograde avec les versions 1.0 et 1.1 de Symfony.

extra_parameters_as_query_string

La valeur par défaut de l'option `extra_parameters_as_query_string` est `true` pour les nouveaux projets et `false` pour les projets mis à jour.

Lorsque des paramètres ne sont pas utilisés dans la génération de la route, la directive de configuration `extra_parameters_as_query_string` autorise ces arguments supplémentaires à être convertis sous la forme d'une chaîne de requête. Placer la valeur à `false` de ce paramètre dans un projet Symfony 1.2 permet de revenir au comportement des version 1.0 et 1.1 de Symfony. En revanche, pour ces dernières, les paramètres supplémentaires seront simplement ignorés par le système de routage.

cache

L'option `cache` définit une factory de cache anonyme à utiliser pour mettre en cache la configuration et les données du système de routage.

suffix

La valeur par défaut de l'option `suffix` est `none`.

L'option `suffix` définit le suffixe à ajouter au bout de chaque route, mais cette fonctionnalité est désormais dépréciée depuis l'intégration du nouveau framework de routage de Symfony 1.2. Par conséquent, cette directive de configuration devient inutile bien qu'elle soit toujours présente.

load_configuration

La valeur par défaut de l'option `load_configuration` est `true`.

L'option `load_configuration` précise si le fichier de configuration `routing.yml` doit être automatiquement chargé et analysé. La valeur `false` doit être fixée pour utiliser le système de routage à l'extérieur d'un projet Symfony.

Le service logger

Configuration par défaut

Le service `logger` est accessible grâce à l'accesseur `getLogger()` de l'objet `sfContext`.

```
sfContext::getInstance()->getLogger()
```

La configuration par défaut du service `logger` est la suivante :

```
logger:
  class: sfAggregateLogger
  param:
    level: debug
    loggers:
      sf_web_debug:
        class: sfWebDebugLogger
        param:
          level: debug
          condition:       %SF_WEB_DEBUG%
          xdebug_logging: true
          web_debug_class: sfWebDebug
```

```
    sf_file_debug:
      class: sfFileLogger
      param:
        level: debug
        file: %SF_LOG_DIR%/%SF_APP%_%SF_ENVIRONMENT%.log
```

Le service `logger` bénéficie également d'une configuration par défaut pour l'environnement de production `prod`.

```
logger:
  class:   sfNoLogger
  param:
    level:   err
    loggers: ~
```

Cette factory est toujours initialisée, mais la procédure d'enregistrement des traces de logs se produit uniquement si le paramètre `logging_enabled` est défini à la valeur `on`.

level

L'option `level` définit le niveau de gravité des informations de logs à enregistrer et prend une valeur parmi `EMERG`, `ALERT`, `CRIT`, `ERR`, `WARNING`, `NOTICE`, `INFO` ou `DEBUG`.

loggers

L'option `loggers` définit une liste des objets à utiliser pour enregistrer les traces de logs. Cette liste est un tableau de factories anonymes d'objets de log qui figurent parmi les classes suivantes : `sfConsoleLogger`, `sfFileLogger`, `sfNoLogger`, `sfStreamLogger` et `sfVarLogger`.

Le service controller

Configuration par défaut

Le service `controller` est accessible grâce à l'accesseur `getController()` de l'objet `sfContext`.

```
sfContext::getInstance()->getController()
```

La configuration par défaut du service `controller` est la suivante :

```
controller:
  class: sfFrontWebController
```

Les services de cache anonymes

Plusieurs factories (`view_cache`, `i18n` et `routing`) tirent partie des avantages de l'objet de cache s'il est défini dans leur configuration respective. La configuration de l'objet de cache est similaire pour toutes les factories. La clé `cache` définit une factory de cache anonyme. Comme n'importe quelle autre factory, elle accepte deux entrées : `class` et `param`. La section `param` peut prendre n'importe quelle option disponible pour la classe de cache.

L'option `prefix` est la plus importante de toutes étant donné qu'elle permet de partager ou de séparer un cache entre différents environnements/applications/projets.

Les classes natives de cache dans Symfony sont `sfAPCCache`, `sfEAcceleratorCache`, `sfFileCache`, `sfMemcacheCache`, `sfNoCache`, `sfSQLiteCache` et `sfXCacheCache`.

Index

R

S

T