



Développez efficacement votre site web avec le framework Symfony2

Par Correcteur
et Alexandre Bacco (winzou)

Ce PDF vous est offert par



Dassault Systèmes recrute : www.3ds.com



Sommaire

Sommaire	2
Partager	5
Développez efficacement votre site web avec le framework Symfony2	7
Partie 1 : Vue d'ensemble de Symfony2	7
Symfony2, un framework PHP	8
Qu'est-ce qu'un framework ?	8
L'objectif d'un framework	8
Pesons le pour et le contre	8
Alors, convaincu ?	9
Qu'est-ce que Symfony2 ?	9
Un framework	9
Un framework populaire	9
Un framework populaire et français	9
Télécharger Symfony2	9
Obtenir Symfony2	9
Vérifier votre configuration de PHP	10
Vérifier l'installation de PHP en console	11
En résumé	12
Vous avez dit Symfony2 ?	12
L'architecture des fichiers	13
Liste des répertoires	13
Le contrôleur frontal	13
L'architecture conceptuelle	14
Architecture MVC	14
Parcours d'une requête dans Symfony2	15
Symfony2 et ses bundles	16
La découpe en « bundles »	17
La structure d'un bundle	17
En résumé	18
Utilisons la console pour créer un bundle	19
Utilisation de la console	19
Sous Windows	19
Sous Linux et Mac	20
À quoi ça sert ?	20
Comment ça marche ?	20
Créons notre bundle	21
Tout est bundle	21
Que s'est-il passé ?	23
À retenir	25
En résumé	25
Partie 2 : Les bases de Symfony2 : « Hello World »	26
Mon premier Hello World avec Symfony2	26
Créons notre route	26
Le routeur (ou « router ») ? Une route ?	26
Créons notre contrôleur	28
Le rôle du contrôleur	28
Créons notre contrôleur	28
Créons notre template Twig	29
Les templates avec Twig	29
Utiliser Twig avec Symfony2	29
Notre objectif : créer un blog	31
Le fil conducteur : un blog	31
Notre blog	31
Un peu de nettoyage	31
Schéma de développement sous Symfony2	31
Pour conclure	32
En résumé	32
Le routeur de Symfony2	33
Le fonctionnement	33
Fonctionnement du routeur	33
Convention pour le nom du contrôleur	35
Les routes de base	35
Créer une route	35
Créer une route avec des paramètres	35
Les routes avancées	37
Créer une route avec des paramètres et leurs contraintes	37
Utiliser des paramètres facultatifs	37
Utiliser des « paramètres système »	38
Ajouter un préfixe lors de l'import de nos routes	38
Générer des URL	38
Pourquoi générer des URL ?	38
Comment générer des URL ?	38
Application : les routes de notre blog	39
Construction des routes	39

Récapitulatif	40
Pour conclure	41
En résumé	41
Les contrôleurs avec Symfony2	41
Le rôle du contrôleur	42
Retourner une réponse	42
Manipuler l'objet Request	42
Les paramètres de la requête	42
Les autres méthodes de l'objet Request	44
Manipuler l'objet Response	44
Décomposition de la construction d'un objet Response	44
Réponses et vues	45
Réponse et redirection	45
Changer le Content-type de la réponse	46
Les différents services	46
Qu'est-ce qu'un service ?	46
Accéder aux services	47
Brève liste des services	48
Application : le contrôleur de notre blog	51
Construction du contrôleur	51
À retenir	52
Testons-le	52
Pour conclure	53
En résumé	53
Le moteur de template Twig	54
Les templates Twig	54
Intérêt	54
En pratique	54
À savoir	55
Afficher des variables	55
Syntaxe de base pour afficher des variables	55
Précisions sur la syntaxe {{ objet.attribut }}	55
Les filtres utiles	56
Twig et la sécurité	56
Les variables globales	56
Structures de contrôle et expressions	57
Les structures de contrôle	57
Les tests utiles	59
Hériter et inclure des templates	59
L'héritage de template	59
Qu'est-ce que l'on vient de faire ?	60
Le modèle « triple héritage »	61
L'inclusion de templates	61
L'inclusion de contrôleurs	62
Application : les templates de notre blog	64
Layout général	64
Layout du bundle	65
Les templates finaux	65
Pour conclure	71
En résumé	71
Installer un bundle grâce à Composer	72
Composer, qu'est-ce que c'est ?	72
Un gestionnaire de dépendances	72
Comment Composer sait où trouver les librairies ?	72
Un outil innovant... dans l'écosystème PHP	72
Concrètement, comment ça marche ?	72
Installer Composer et Git	72
Installer Composer	72
Installer Git	73
Installer un bundle grâce à Composer	73
Manipulons Composer	73
Mettions à jour Symfony2	74
Installer un bundle avec Composer	75
Gérer l'autoload d'une bibliothèque manuellement	76
Pour conclure	77
En résumé	77
Partie 3 : Gérer la base de données avec Doctrine2	77
La couche métier : les entités	78
Notions d'ORM : fini les requêtes, utilisons des objets	78
Vos données sont des objets	78
Créer une première entité avec Doctrine2	78
Une entité, c'est juste un objet	78
Une entité, c'est juste un objet... mais avec des commentaires !	79
Générer une entité : le générateur à la rescousse !	80
Affiner notre entité avec de la logique métier	82
Conclusion	83
Tout sur le mapping !	83
L'annotation Entity	84
L'annotation Table	84
L'annotation Column	84
Pour conclure	85
En résumé	86

Manipuler ses entités avec Doctrine2	86
Matérialiser les tables en base de données	87
Créer la table correspondante dans la base de données	87
Modifier une entité	88
À retenir	89
Enregistrer ses entités avec l'EntityManager	89
Les services Doctrine2	89
Enregistrer ses entités en base de données	91
Récupérer ses entités avec un EntityRepository	94
En résumé	95
Les relations entre entités avec Doctrine2	95
Présentation	96
Présentation	96
Les différents types de relations	96
Notions techniques d'ORM à savoir	96
Rien n'est magique	96
Relation One-To-One	97
Présentation	97
Définition de la relation dans les entités	98
Exemple d'utilisation	100
Relation Many-To-One	101
Présentation	102
Définition de la relation dans les entités	104
Exemple d'utilisation	105
Relation Many-To-Many	106
Présentation	106
Définition de la relation dans les entités	108
Remplissons la base de données avec les Fixtures	110
Exemples d'utilisation	111
Relation Many-To-Many avec attributs	113
Présentation	113
Définition de la relation dans les entités	114
Remplissons la base de données	116
Exemple d'utilisation	117
Les relations bidirectionnelles	119
Présentation	119
Définition de la relation dans les entités	119
Pour conclure	122
En résumé	122
Récupérer ses entités avec Doctrine2	123
Le rôle des Repository	123
Définition	123
Les méthodes de récupération des entités	123
Les méthodes de récupération de base	123
Définition	123
Les méthodes normales	123
Les méthodes magiques	125
Les méthodes de récupération personnelles	125
La théorie	125
Utilisation du Doctrine Query Language (DQL)	130
Utiliser les jointures dans nos requêtes	132
Pourquoi utiliser les jointures ?	132
Comment faire des jointures avec le QueryBuilder ?	132
Comment utiliser les jointures ?	133
Application : les entités de notre blog	133
Plan d'attaque	133
À vous de jouer !	133
Le code	133
En résumé	134
Les événements et extensions Doctrine	135
Les événements Doctrine	135
L'intérêt des événements Doctrine	135
Définir des callbacks de cycle de vie	135
Liste des événements de cycle de vie	136
Les extensions Doctrine	137
L'intérêt des extensions Doctrine	137
Installer le StofDoctrineExtensionBundle	137
Utiliser une extension : l'exemple de Sluggable	137
Liste des extensions Doctrine	139
Pour conclure	139
En résumé	139
TP : Les entités de notre blog	139
Synthèse des entités	140
Entité Article	140
Entité Image	143
Entité Commentaire	145
Entité Catégorie	146
Entités Compétence et ArticleCompetence	147
Et bien sûr...	149
Adaptation du contrôleur	149
Théorie	149
Pratique	149

Amélioration du contrôleur	151
L'utilisation d'un ParamConverter	151
Utiliser une jointure pour récupérer les articles	152
La pagination des articles sur la page d'accueil	154
Améliorer les vues	156
Pour conclure	157
En résumé	157
Partie 4 : Allons plus loin avec Symfony2	157
Créer des formulaires avec Symfony2	158
Gestion des formulaires	158
L'enjeu des formulaires	158
Un formulaire Symfony2, qu'est-ce que c'est ?	158
Gestion basique d'un formulaire	159
Gestion de la soumission d'un formulaire	161
Gérer les valeurs par défaut du formulaire	163
Personnaliser l'affichage d'un formulaire	163
Créer des types de champs personnalisés	164
Externaliser la définition de ses formulaires	164
Définition du formulaire dans ArticleType	164
Le contrôleur épuré	165
Les formulaires imbriqués	166
Intérêts de l'imbrication	166
Un formulaire est un champ	166
Relation simple : imbriquer un seul formulaire	167
Relation multiple : imbriquer un même formulaire plusieurs fois	169
Validez vos données	171
Pourquoi valider des données ?	172
Never Trust User Input	172
L'intérêt de la validation	172
La théorie de la validation	172
Définir les règles de validation	172
Les différentes formes de règles	172
Définir les règles de validation	172
Créer ses propres contraintes	175
Déclencher la validation	175
Le service Validator	175
La validation automatique sur les formulaires	176
Conclusion	176
Encore plus de règles de validation	176
Valider depuis un Getter	176
Valider intelligemment un attribut objet	177
Valider depuis un Callback	178
Valider un champ unique	178
Valider selon nos propres contraintes	179
1. Créer la contrainte	179
2. Créer le validateur	180
3. Transformer son validateur en service	181
Pour conclure	182
En résumé	183
Sécurité et gestion des utilisateurs	184
Authentification et autorisation	184
Les notions d'authentification et d'autorisation	184
Exemples	184
Les différents acteurs de la sécurité Symfony2	186
Le fichier de configuration de la sécurité	186
Les acteurs de l'authentification	186
Les acteurs de l'autorisation	187
Installation du bundle FOSUserBundle	188
Télécharger FOSUserBundle	188
Activer le bundle	188
Personnaliser FOSUserBundle	188
Configuration de la sécurité avec FOSUserBundle	190
Configuration de la sécurité	190
Configuration du bundle FOSUserBundle	191
Tester l'authentification	192
Personnalisation esthétique du bundle	192
Gestion des autorisations avec les rôles	194
Définition des rôles	194
Tester les rôles de l'utilisateur	194
Gérer les rôles des utilisateurs	195
Manipuler les utilisateurs	196
Manipuler les utilisateurs	196
Récupérer l'utilisateur courant	196
Les services, théorie et création	198
Qu'est ce qu'un service ?	198
La persistance des services	198
Le service container	198
En pratique	198
Créer un service	199
Création de la classe du service	199
Création de la configuration du service	201
Utilisation du service	202

Les services, utilisation poussée	203
Les arguments	203
Les calls	204
Les tags	205
Extension Twig	205
Les évènements du cœur	206
Les types de champs de formulaire	206
Les autres tags	207
Les services courants de Symfony2	208
Les services courants de Symfony	208
Partie 5 : Astuces et points particuliers	211
Récupérer directement des entités Doctrine dans son contrôleur	211
Théorie : pourquoi un ParamConverter ?	211
Récupérer des objets Doctrine avant même le contrôleur	211
Les ParamConverters	211
Un ParamConverter utile : DoctrineParamConverter	211
Un peu de théorie sur les ParamConverter	211
Pratique : utilisation de DoctrineParamConverter	211
Utiliser DoctrineParamConverter	211
Personnaliser les pages d'erreur	212
Théorie : remplacer les templates d'un bundle	213
Pratique : remplacer les templates Exception de TwigBundle	213
Le contenu d'une page d'erreur	213
Utiliser la console directement depuis le navigateur !	215
Théorie : Le composant Console de Symfony2	215
Les commandes sont en PHP	215
Exemple d'une commande	215
Pratique : Utiliser un ConsoleBundle	216
ConsoleBundle ?	216
Installer CoreSphereConsoleBundle	217
Utilisation de la console dans son navigateur	219
Prêt pour l'hébergement mutualisé	219
Déployer son site Symfony2 en production	220
Préparer son application en local	220
Vider le cache, tout le cache	220
Tester l'environnement de production	220
Soigner ses pages d'erreurs	220
Installer une console sur navigateur	220
Vérifier et préparer le serveur de production	221
Vérifier la compatibilité du serveur	221
Modifier les paramètres OVH pour être compatible	222
Déployer votre application	222
Envoyer les fichiers sur le serveur	222
Régler les droits sur les dossiers app/cache et app/logs	222
S'autoriser l'environnement de développement	222
Mettre en place la base de données	223
S'assurer que tout fonctionne	223
Avoir de belles URL	223
Et profitez !	224
Avancement du cours	224
En attendant	224
Licences	224



Développez efficacement votre site web avec le framework Symfony2



Le tutoriel que vous êtes en train de lire est en **bêta-test**. Son auteur souhaite que vous lui fassiez part de vos commentaires pour l'aider à l'améliorer avant sa publication officielle. Notez que le contenu n'a pas été validé par l'équipe éditoriale du Site du Zéro.



Par

Alexandre Bacco (winzou)

Mise à jour : 08/01/2013

Difficulté : Intermédiaire

Durée d'étude : 1 mois



20 253 visites depuis 7 jours, classé 15/800

Vous savez déjà faire des sites Internet ? Vous maîtrisez votre code, mais n'êtes pas totalement satisfait ? Vous avez trop souvent l'impression de réinventer la roue ?

Alors ce tutoriel est fait pour vous !

Symfony2 est un puissant framework qui va vous permettre de réaliser des sites complexes rapidement, mais de façon structurée et avec un code clair et maintenable. En un mot : le paradis du développeur !

Ce tutoriel est un tutoriel pour débutants sur Symfony2, vous n'avez besoin d'aucune notion sur les frameworks pour l'aborder, nous allons les découvrir ensemble. Cependant, il est fortement conseillé :

- d'avoir déjà une bonne expérience de PHP ([Aller au cours Concevez votre site web avec PHP et MySQL](#)) ;
- de maîtriser les notions de base de la POO ([Aller au cours La programmation orientée objet](#)) ;
- d'avoir éventuellement des notions de namespace ([Aller au cours Les espaces de nom](#)).



Si vous ne maîtrisez pas ces trois points, je vous invite vraiment à les apprendre avant de commencer la lecture de ce cours. Symfony2 requiert ces bases, et si vous ne les avez pas, vous risquez de mettre plus de temps pour assimiler ce tutoriel. C'est comme acheter un A380 sans savoir piloter : c'est joli mais ça ne sert à rien.

Partie 1 : Vue d'ensemble de Symfony2

Pour débuter, quoi de mieux que de commencer par le commencement ! Si vous n'avez aucune expérience dans les *frameworks* ni dans l'architecture MVC, cette partie sera très riche en nouvelles notions. Avançons doucement mais sûrement, vous êtes là pour apprendre !

Symfony2, un framework PHP

Alors, vous avez décidé de vous lancer dans Symfony2 ? Parfait, vous ne le regretterez pas ! Tout au long de ce cours, nous apprendrons à utiliser ce framework, et vous comprendrez petit à petit la puissance de cet outil. Mais tout d'abord, commençons par les bases, et voyons précisément quels sont les objectifs et les limites d'un framework tel que Symfony2.

Dans ce chapitre nous allons découvrir pourquoi Symfony2 est un bon choix pour votre application web. Une boîte à outils faite en PHP qui a pour but de vous simplifier la vie, c'est toujours sympa non ? Allons-y !

Qu'est-ce qu'un framework ?

L'objectif d'un framework

L'objectif de ce chapitre n'est pas de vous fournir toutes les clés pour *concevoir* un framework, mais suffisamment pour pouvoir en *utiliser* un. On exposera rapidement les intérêts, les avantages et les inconvénients de l'utilisation d'un tel outil.

Définition

Le mot « *framework* » provient de l'anglais « *frame* » qui veut dire « cadre » en français, et « *work* » qui signifie « travail ». Littéralement, c'est donc un « cadre de travail ». Vous voilà avancé, hein ? 😊 Concrètement, c'est un ensemble de composants qui servent à créer les fondations, l'architecture et les grandes lignes d'un logiciel. Il existe des centaines de frameworks couvrant la plupart des langages de programmation. Ils sont destinés au développement de sites Web ou bien à la conception de logiciels.

Un framework est une boîte à outils conçue par un ou plusieurs développeurs à destination d'autres développeurs. Contrairement à certains scripts tels que Wordpress, DotClear ou autres, un framework n'est pas utilisable tel quel. Il n'est pas fait pour être utilisé par les utilisateurs finaux. Le développeur qui se sert d'un framework a encore du boulot à fournir, d'où ce cours !

Objectif d'un framework

L'objectif premier d'un framework est d'améliorer la productivité des développeurs qui l'utilisent. Plutôt sympa non ? Souvent organisé en différents composants, un framework offre la possibilité au développeur final d'utiliser tel ou tel composant pour lui faciliter son développement, et lui permet ainsi de se concentrer sur le plus important.

Prenons un exemple concret. Il existe dans Symfony2 un composant qui gère les formulaires HTML : leur affichage, leur validation, etc. Le développeur qui l'utilise se concentre sur l'essentiel dans son application : chaque formulaire effectue une action, et c'est cette action qui est importante, pas les formulaires. Étendez ce principe à toute une application ou tout un site internet, et vous comprenez l'intérêt d'un framework ! Autrement dit, le framework s'occupe de la forme et permet au développeur de se concentrer sur le fond.

Pesons le pour et le contre

Comme tout bon développeur, lorsqu'on veut utiliser un nouvel outil, on doit en peser le pour et le contre pour être sûr de faire le bon choix !

Les pour

L'avantage premier est donc, on vient de le voir, le gain en productivité. Mais il en existe bien d'autres ! On peut les classer en plusieurs catégories : le code, le travail et la communauté.

Tout d'abord, un framework va vous aider à réaliser un « bon code ». Par bon code j'entends qu'il vous incite, de part sa propre architecture, à bien organiser votre code. Et un code bien organisé est un code facilement maintenable et évolutif ! De plus, un framework offre des briques prêtes à être utilisées (le composant *Formulaire* de Symfony2 par exemple), ce qui vous évite de réinventer la roue, et surtout qui vous permet d'utiliser des briques puissantes et éprouvées. En effet ces briques sont développées par des équipes de développeurs chevronnés, elles sont donc très flexibles et très robustes. Vous économisez ainsi des heures de développement !

Ensuite, un framework améliore la façon dont vous travaillez. En effet, dans le cas d'un site internet vous travaillez souvent avec d'autres développeurs PHP et un designer. Un framework vous aide doublement dans ce travail en équipe. D'une part, un framework utilise presque toujours l'architecture MVC ; on en reparlera mais sachez pour le moment que c'est une façon d'organiser son code qui sépare le code PHP du code HTML. Ainsi votre designer peut travailler sur des fichiers différents des vôtres, fini les problèmes d'édition simultanée d'un même fichier ! D'autre part, un framework a une structure et des conventions de codes connues. Ainsi vous pouvez facilement recruter un autre développeur : s'il connaît déjà le framework en question, il s'intègrera très rapidement au projet.

Enfin, le dernier avantage est la communauté soutenant chaque framework. C'est elle qui fournit les tutoriaux ou les cours (comme celui que vous lisez !), de l'aide sur les forums, et bien sûr les mises à jour du framework. Ces mises à jour sont très importantes : imaginez que vous codiez vous-même tout ce qui est connexion utilisateur, session, moteur de *templates*, etc. Comme il est impossible de coder sans *bugs*, vous devriez logiquement corriger chaque *bug* déclaré sur votre code. Maintenant imaginez que toutes ces briques de votre site, qui ne sont pas forcément votre tasse de thé, soient fournies par le framework. À chaque fois que vous ou les milliers d'autres utilisateurs du framework trouverez un *bug*, les développeurs et la communauté s'occupent de le corriger, et vous n'aurez plus qu'à suivre les mises à jour. Un vrai paradis !

Il existe plein d'autres avantages que je ne vais pas vous détailler, mais un framework c'est aussi :

- Une communauté active qui utilise le framework et qui contribue en retour ;
- Une documentation de qualité et régulièrement mise à jour ;
- Un code source maintenu par des développeurs attitrés ;
- Un code qui respecte les standards de programmation ;
- Un support à long terme garanti et des mises à jour qui ne cassent pas la compatibilité ;
- Etc.

Les contre

Vous vous en doutez, avec autant d'avantages il y a forcément des inconvénients. Et bien figurez-vous qu'il n'y en a pas tant que ça !

S'il ne fallait en citer qu'un, cela serait évidemment la courbe d'apprentissage qui est plus élevée. En effet, pour maîtriser un framework il faut un temps d'apprentissage non négligeable. Chaque brique qui compose un framework a sa complexité propre qu'il vous faudra apprêhender.

Notez également que pour les frameworks les plus récents, tel que Symfony2 justement, il faut également être au courant des dernières nouveautés de PHP. Je pense notamment à la [Programmation Orientée Objet](#) et aux [namespaces](#). De plus, connaître certaines bonnes pratiques telle que l'architecture MVC est un plus.

Mais, rien de tout cela ne doit vous effrayer ! Vöyez l'apprentissage d'un framework comme un investissement : il y a un certain effort à fournir au début, mais les résultats se récoltent ensuite sur le long terme !

Alors, convaincu ?

J'espère vous avoir convaincu que les pour l'emportent largement sur les contre. Si vous êtes prêt à relever le défi aujourd'hui pour être plus productif demain, alors ce cours est fait pour vous !

Qu'est-ce que Symfony2 ?

Un framework

Symfony2 est donc un framework PHP. Bien sûr il en existe d'autres, pour ne citer que les plus connus : [Zend Framework](#), [CodeIgniter](#), [CakePHP](#), etc. Le choix d'un framework est assez personnel, et doit être adapté au projet engagé. Sans vouloir prêcher pour ma paroisse, Symfony2 est l'un des plus flexibles et des plus puissants.

Un framework populaire

Symfony est très populaire. C'est un des frameworks les plus utilisés dans le monde, notamment dans les entreprises. Il est utilisé par Dailymotion par exemple ! La première version de symfony est sortie en 2005 et est aujourd'hui toujours très utilisée. Cela lui apporte un retour d'expérience et une notoriété exceptionnelle. Aujourd'hui, beaucoup d'entreprises dans le domaine de l'Internet (dont Simple IT, l'éditeur du Site du Zéro !) recrutent des développeurs capables de travailler sous ce framework. Ces développeurs pourront ainsi se greffer aux projets de l'entreprise très rapidement, car ils en connaîtront déjà les grandes lignes. C'est un atout si vous souhaitez travailler dans ce domaine. 😊

La deuxième version, que nous étudierons dans ce tutoriel, est sortie en août 2011. Encore jeune, son développement a été fulgurant grâce à une communauté de développeurs dévoués. Bien que différent dans sa conception, cette deuxième version est plus rapide et plus souple que la première. Il y a fort à parier que très rapidement, beaucoup d'entreprises s'arracheront les compétences des premiers développeurs Symfony2. Faites-en partie !

Un framework populaire et français

Eh oui, Symfony2, l'un des meilleurs frameworks PHP au monde, est un framework français ! Il est édité par la société [SensioLabs](#), dont le créateur est Fabien Potencier. Mais Symfony2 étant un script open source, il a également été écrit par toute la communauté : beaucoup de français, mais aussi des développeurs de tout horizon : Europe, États-Unis, etc. C'est grâce au talent de Fabien et à la générosité de la communauté que Symfony2 a vu le jour.

Télécharger Symfony2

Obtenir Symfony2

Il existe de nombreux moyens d'obtenir Symfony2. Nous allons voir ici la méthode la plus simple : télécharger la distribution standard. Pour cela rien de plus simple, rendez-vous sur le site de Symfony2, rubrique [Download](#), et téléchargez la version « Symfony Standard (.zip) ». Si vous utilisez déjà Composer, vous pouvez télécharger cette distribution standard en une seule commande :

Code : Console

```
php composer.phar create-project symfony/framework-standard-edition Symfony 2.1.3
```

Cela va installer la version 2.1 de Symfony2 dans le répertoire `Symfony`. Pour les autres, téléchargez l'archive traditionnelle avec

les *vendors*, on parlera de Composer dans un prochain chapitre.



Si vous êtes sous Windows, évitez de télécharger l'archive au format .tgz car des problèmes ont été rencontrés avec cette archive.

Une fois l'archive téléchargée, décompressez les fichiers dans votre répertoire web habituel, par exemple C:\wamp\www pour Windows ou /var/www pour Linux. Pour la suite du tutoriel, je considérerai que les fichiers sont accessibles à l'URL <http://localhost/Symfony>. Je vous recommande d'avoir la même adresse, car je ferai ce genre de liens tout au long du tutoriel. 😊

Vérifier votre configuration de PHP

Symfony2 a quelques contraintes par rapport à votre configuration PHP. Par exemple, il ne tourne que sur la version 5.3.2 ou supérieure de PHP. Pour vérifier si votre environnement est compatible, rendez-vous à l'adresse suivante : <http://localhost/Symfony/web/config.php>. Si vous avez une version adéquate de PHP, vous devriez obtenir la figure suivante.

Mon environnement de travail est compatible avec Symfony2 !

En cas d'incompatibilité (version de PHP notamment), Symfony2 vous demande de régler les problèmes avant de continuer. S'il ne vous propose que des « recommandations », vous pouvez continuer sans problème. Ce sont des points que je vous conseille de régler, mais qui sont facultatifs.

Si vous êtes sous Linux, vous devez bien régler les droits sur les répertoires app/cache et app/logs afin que Symfony2 puisse y écrire. Placez-vous dans le répertoire Symfony et videz d'abord ces répertoires :

Code : Console

```
rm -rf app/cache/*
rm -rf app/logs/*
```

Ensuite, si votre distribution supporte le chmod +a exécutez ces commandes pour définir les bons droits :

Code : Console

```
sudo chmod +a "www-data allow delete,write,append,file_inherit,directory_inherit" a
sudo chmod +a "`whoami` allow delete,write,append,file_inherit,directory_inherit" a
```

Si vous rencontrez une erreur avec ces commandes, exécutez les commandes suivantes, qui n'utilisent pas le chmod +a :

Code : Console

```
sudo setfacl -R -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
```

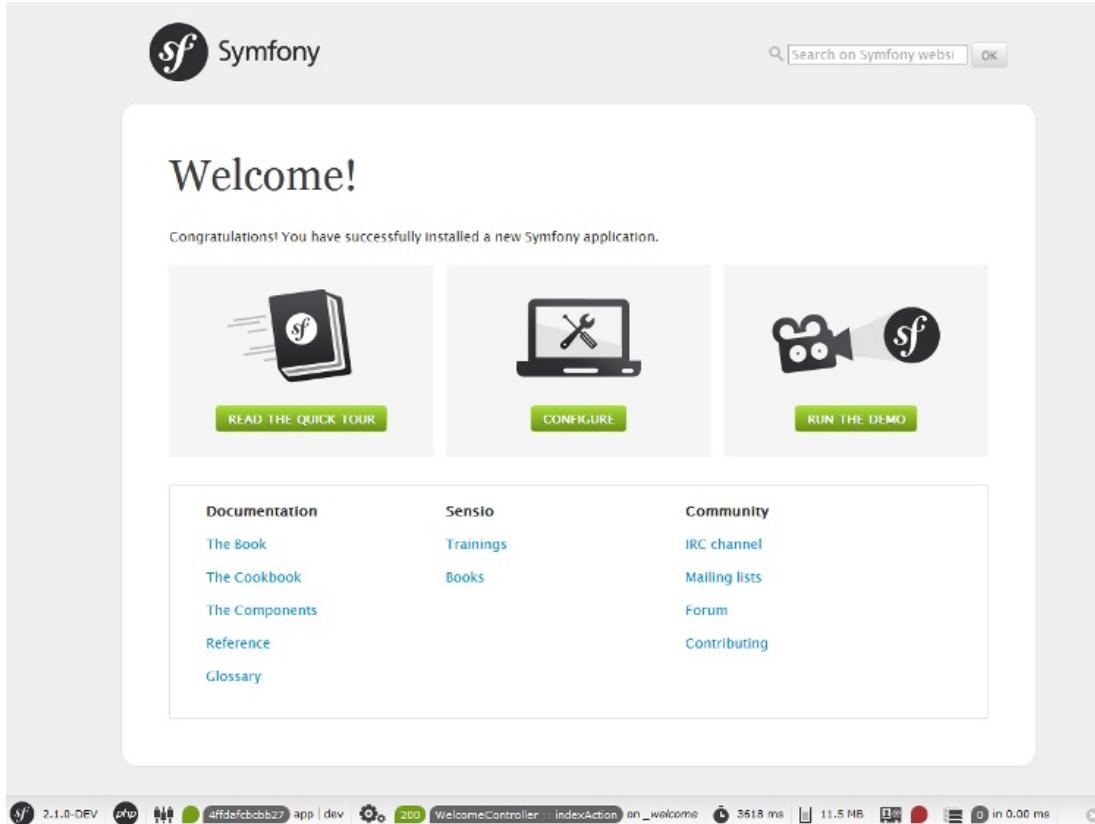
```
sudo setfacl -dR -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
```

Enfin, si vous ne pouvez pas utiliser les ACL (utilisés dans les commandes précédentes), définissez simplement les droits comme suit :

Code : Console

```
chmod 777 app/cache
chmod 777 app/logs
```

Vous pouvez dès à présent exécuter Symfony2, félicitations ! Rendez-vous sur la page http://localhost/Symfony/web/app_dev.php, vous devriez avoir quelque chose ressemblant à la figure suivante.



La page d'accueil de Symfony2

Vérifier l'installation de PHP en console

Nous aurons parfois besoin d'exécuter des commandes PHP via la console, pour générer du code ou gérer la base de données. Ce sont des commandes qui vont nous faire gagner du temps (toujours le même objectif !), vérifions donc que PHP est bien disponible en console.

Si vous êtes sous Linux vous ne devriez pas avoir de soucis, PHP est bien disponible en console. Si vous êtes sous Windows, rien n'est sûr. Dans tous les cas, vérifiez-le en ouvrant l'invite de commandes pour Windows, ou le terminal pour Linux, et entrez la commande suivante : `php -v`. Si cette commande vous retourne bien la version de PHP et d'autres informations, c'est tout bon pour vous.

La commande vous affiche une erreur ?

Si vous êtes sous Windows, PHP est bien installé mais Windows ne sait pas où le trouver, il faut juste le lui dire. Voici la démarche à suivre pour régler ce problème :

1. Allez dans les paramètres systèmes avancés (Démarrer > Panneau de configuration > Système et sécurité > Système > Paramètres système avancés);
2. Cliquez sur le bouton Variables d'environnements...;
3. Regardez dans le panneau Variables systèmes;
4. Trouvez l'entrée Path (vous devriez avoir à faire descendre l'ascenseur pour le trouver);
5. Double cliquez sur l'entrée Path;
6. Entrez votre répertoire PHP à la fin, sans oublier le point virgule (;) avant. C'est le répertoire dans lequel se trouve le fichier php.exe. Par exemple ;C:\wamp\bin\php\php5.3;

7. Confirmez en cliquant sur OK

Vous devez ensuite redémarrer l'invite de commande pour prendre en compte les changements.

Si vous êtes sous Linux, vérifiez votre installation de PHP. Vous devez notamment avoir le paquet `php5-cli`, qui est la version console de PHP.

Dans les deux cas, vérifiez après vos manipulations que le problème est bien résolu. Pour cela, exécutez à nouveau la commande `php -v`. Elle devrait alors vous afficher la version de PHP.

Et voilà, votre poste de travail est maintenant opérationnel pour développer avec Symfony2 !

En résumé

- Le mot « framework » signifie « cadre de travail » en français ;
- L'objectif principal d'un framework est d'améliorer la productivité des développeurs qui l'utilisent ;
- Contrairement aux CMS, un framework est destiné à des développeurs, et non à des novices en informatique ;
- L'apprentissage d'un framework est un investissement : il y a un certain effort à fournir au début, mais les résultats se récoltent ensuite sur le long terme !
- Symfony2 est un framework PHP très populaire, français, et très utilisé dans le milieu des entreprises.

Vous avez dit Symfony2 ?

Dans ce chapitre, nous allons voir comment est organisé Symfony2 à l'intérieur. Nous n'entrerons pas dans les détails, c'est trop tôt, le but est juste d'avoir une vision globale du processus d'exécution d'une page sous Symfony2. Ainsi, vous pourrez comprendre ce que vous faites. C'est mieux, non ? 😊

L'architecture des fichiers

On vient d'extraire beaucoup de fichiers, mais sans savoir encore à quoi ils servent. C'est le moment d'éclaircir tout cela !

Liste des répertoires

Ouvrez donc le répertoire dans lequel vous avez extrait les fichiers. Vous pouvez voir qu'il n'y a pas beaucoup de fichiers ici, seulement des répertoires. En effet tout est bien rangé dans chaque répertoire, il nous faut donc comprendre à quoi ils servent. En voici la liste :

- app
- src
- vendor
- web

Le répertoire /app

Ce répertoire contient tout ce qui concerne votre site internet... sauf son code source. Assez étrange me direz vous. En fait c'est simplement pour séparer le code source qui fait la logique de votre site du reste. Le reste, c'est ce répertoire app. Et ce reste c'est : la configuration, le cache, les fichiers logs, etc. Ce sont des fichiers qui concernent l'entièreté de votre site, contrairement aux fichiers de code source qui seront découpés par fonctionnalités de votre site. Dans Symfony2, un projet de site internet est une **application**, simple question de vocabulaire. Le répertoire /app est donc le raccourci pour « application ».

Le répertoire /src

Voilà enfin le répertoire dans lequel on mettra le code source ! C'est ici que l'on passera le plus clair de notre temps. Dans ce répertoire, nous organiserons notre code en *bundles*, des briques de notre application, dont nous verrons la définition plus loin.

Vous pouvez voir que ce répertoire n'est pas vide : il contient en effet quelques fichiers exemples, fournis par Symfony2. Nous les supprimerons plus tard dans ce cours.

Le répertoire /vendor

Ce répertoire contient toutes les bibliothèques externes à notre application. Dans ces bibliothèques externes, j'inclus Symfony2 ! Vous pouvez parcourir ce répertoire, vous y trouverez des bibliothèques comme Doctrine, Twig, SwiftMailer, etc.



Et une bibliothèque, c'est quoi exactement ?

Une bibliothèque est une sorte de boîte noire qui remplit une fonction bien précise, et dont on peut se servir dans notre code. Par exemple, la bibliothèque SwiftMailer permet d'envoyer des emails. On ne sait pas comment elle fonctionne (principe de la boîte noire), mais on sait comment s'en servir : on pourra donc envoyer des emails très facilement, juste en apprenant rapidement à utiliser la bibliothèque.

Le répertoire /web

Ce répertoire contient tous les fichiers destinés à vos visiteurs : images, fichiers CSS et JavaScript, etc. Il contient également le contrôleur frontal (`app.php`), dont nous parlerons juste après.

En fait c'est le seul répertoire qui devrait être accessible à vos visiteurs. Les autres répertoires ne sont pas censés être accessibles (ce sont vos classes, elles vous regardent vous, pas vos visiteurs), c'est pourquoi vous y trouverez des fichiers `.htaccess` interdisant l'accès depuis l'extérieur. On utilisera donc toujours des URL du type `http://localhost/Symfony/web/...` au lieu de simplement `http://localhost/Symfony/...`.

 Si vous le souhaitez, vous pouvez configurer votre Apache pour que l'URL `http://localhost/Symfony` pointe directement sur le répertoire `web/`. Pour cela, vous pouvez lire ce [tutoriel](#) qui explique comment configurer Apache. Cependant ce n'est pas très important pour le développement, on en reparlera plus loin.

À retenir

Retenez donc que nous passerons la plupart de notre temps dans le répertoire `/src`, à travailler sur nos *bundles*. On touchera également pas mal au répertoire `/app` pour configurer notre application. Et lorsque nous installerons des *bundles* téléchargés, nous le ferons dans le répertoire `/vendor`.

Le contrôleur frontal

Définition

Le contrôleur frontal (*front controller*, en anglais) est le point d'entrée de votre application. C'est *le* fichier par lequel passent toutes vos pages. Vous devez sûrement connaître le principe du `index.php` et des pseudo-frames (avec des URL du type `index.php?page=blog`) ; et bien ce `index.php` est un contrôleur frontal. Dans Symfony2, le contrôleur frontal se situe dans le répertoire `/web`, il s'agit de `app.php` ou `app_dev.php`.



Pourquoi y a-t-il deux contrôleurs frontaux ? Normalement c'est un fichier unique qui gère toutes les pages non ?

Vous avez parfaitement raison... pour un code classique ! Mais nous travaillons maintenant avec Symfony2, et son objectif est de nous faciliter le développement. C'est pourquoi Symfony2 propose un contrôleur frontal pour nos visiteurs, `app.php`, et un contrôleur frontal lorsque nous développons, `app_dev.php`. Ces deux contrôleurs frontaux, fournis par Symfony2 et prêts à l'emploi, définissent en fait deux environnements de travail.

Deux environnements de travail

L'objectif est de répondre au mieux suivant la personne qui visite le site :

- Un développeur a besoin d'informations sur la page afin de l'aider à développer. En cas d'erreur, il veut tous les détails pour pouvoir déboguer facilement. Il n'a pas besoin de rapidité.
- Un visiteur normal n'a pas besoin d'informations particulières sur la page. En cas d'erreur, l'origine de celle-ci ne l'intéresse pas du tout, il veut juste retourner d'où il vient. Par contre, il veut que le site soit le plus rapide possible à charger.

Vous voyez la différence ? À chacun ses besoins, et Symfony2 compte bien tous les remplir. C'est pourquoi il offre plusieurs environnements de travail :

- L'environnement de développement, appelé « dev », accessible en utilisant le contrôleur frontal `app_dev.php`. C'est l'environnement que l'on utilisera *toujours* pour développer.
- L'environnement de production, appelé « prod », accessible en utilisant le contrôleur frontal `app.php`.

Essayez-les ! Allez sur http://localhost/Symfony/web/app_dev.php et vous verrez une barre d'outils en bas de votre écran, contenant nombre d'informations utiles au développement. Allez sur <http://localhost/Symfony/web/app.php> et vous obtiendrez... une erreur 404. 😊 En effet, aucune page n'est définie par défaut pour le mode « prod ». Nous les définirons plus tard, mais notez que c'est une « belle » erreur 404, aucun terme barbare n'est employé pour la justifier. 😊

Pour voir le comportement du mode « dev » en cas d'erreur, essayez aussi d'aller sur une page qui n'existe pas. Vous avez vu ce que donne une page introuvable en mode « prod », mais allez maintenant sur http://localhost/Symfony/web/app_dev.php [...] *quinxistepas*. La différence est claire : le mode « prod » nous dit juste « page introuvable » alors que le mode « dev » nous donne plein d'informations sur l'origine de l'erreur, indispensables pour la corriger.

C'est pourquoi, dans la suite du tutoriel, nous utiliserons toujours le mode « dev », en passant donc par `app_dev.php`. Bien sûr, lorsque votre site sera opérationnel et que des internautes pourront le visiter, il faudra leur faire utiliser le mode « prod ». Mais nous n'en sommes pas encore là.



Et comment savoir quelles erreurs surviennent en mode production si elles ne s'affichent pas ?

C'est une bonne question, en effet si par malheur une erreur intervient pour l'un de vos visiteurs, il ne verra aucun message et vous non plus, une vraie galère pour déboguer ! En réalité si les erreurs ne sont pas affichées, elles sont bien stockées dans un fichier. Allez jeter un œil au fichier `app/logs/prod.log` qui contient plein d'informations sur les requêtes effectuées en mode production, dont les erreurs.

Concrètement, qu'est-ce que contrôle le contrôleur frontal ?

Très bonne question. Pour cela, rien de tel que... d'ouvrir le fichier `app.php`. Ouvrez-le et vous constaterez qu'il ne fait pas grand chose. En effet, le but du contrôleur frontal n'est pas de *faire* quelque chose, mais d'être un *point d'entrée* de notre application. Il se limite donc à appeler le noyau (Kernel) de Symfony2 en disant « *On vient de recevoir une requête, transforme-la en réponse s'il-te-plait* ».

Ici, voyez le contrôleur frontal comme un fichier à nous (il est dans notre répertoire `/web`), et le Kernel comme un composant Symfony2, une boîte noire (il est dans le répertoire `/vendor`). Vous voyez comment on a utilisé notre premier composant Symfony2 : on a délégué la gestion de la requête au Kernel. Bien sûr ce Kernel aura besoin de nous pour savoir quoi exécuter comme code, mais il gère déjà plusieurs choses que nous avons vues : la gestion des erreurs, l'ajout de la `toolbar` en bas de l'écran, etc. On n'a encore rien fait, et pourtant on a déjà gagné du temps !

L'architecture conceptuelle

On vient de voir comment sont organisés les fichiers de Symfony2. Maintenant il s'agit de comprendre comment s'organise l'exécution du code au sein de Symfony2.

Architecture MVC

Vous avez certainement déjà entendu parler de ce concept. Sachez que Symfony2 respecte bien entendu cette architecture MVC. Je ne vais pas rentrer dans ses détails car il y a déjà un [super cours sur le Site du Zéro](#), mais en voici les grandes lignes.

MVC signifie « Modèle / Vue / Contrôleur ». C'est un découpage très répandu pour développer les sites Internet, car il sépare les

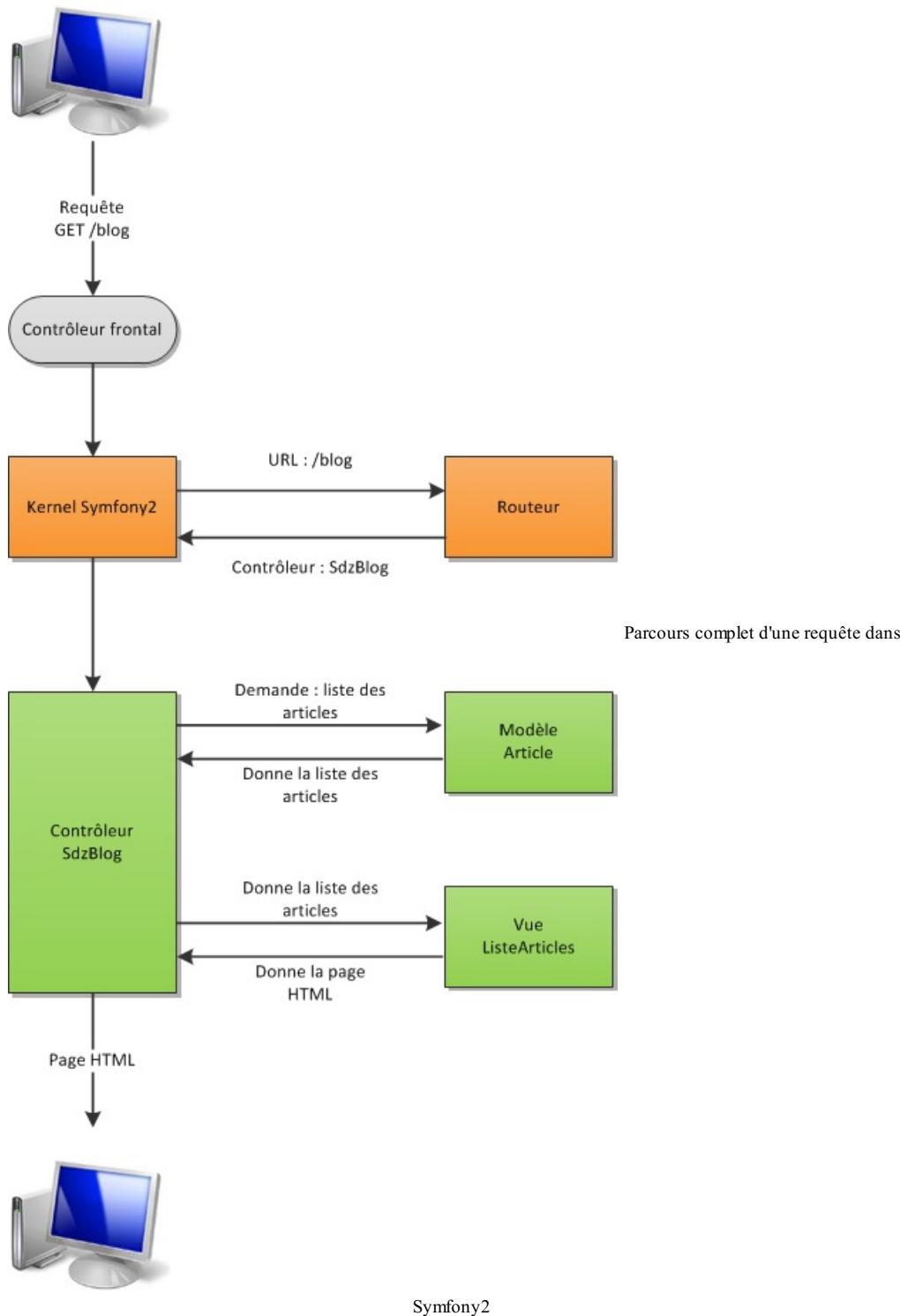
couches selon leur logique propre :

- Le **Contrôleur** (ou *Controller*) : son rôle est de générer la réponse à la requête HTTP demandée par notre visiteur. Il est la couche qui se charge d'analyser et de traiter la requête de l'utilisateur. Le contrôleur contient la logique de notre site Internet et va se contenter « d'utiliser » les autres composants : les modèles et les vues. Concrètement un contrôleur va récupérer, par exemple, les informations sur l'utilisateur courant, vérifier qu'il a le droit de modifier tel article, récupérer cet article et demander la page du formulaire d'édition de l'article. C'est tout bête, avec quelques `if()`, on s'en sort très bien ;
- Le **Modèle** (ou *Model*) : son rôle est de gérer vos données et votre contenu. Reprenons l'exemple de l'article. Lorsque je dis « le contrôleur récupère l'article », il va en fait faire appel au modèle « article » et lui dire : « donne-moi l'article portant l'id 5 ». C'est le modèle qui sait comment récupérer cet article, généralement *via* une requête au serveur SQL, mais ça pourrait être depuis un fichier texte ou ce que vous voulez. Au final, il permet au contrôleur de manipuler les articles, mais sans savoir comment les articles sont stockés, gérés, etc. C'est une couche d'abstraction ;
- La **Vue** (ou *View*) : son rôle est d'afficher les pages. Reprenons encore l'exemple de l'article. Ce n'est pas le contrôleur qui affiche le formulaire, il ne fait qu'appeler la bonne vue. Si nous avons une vue « formulaire », les balises HTML du formulaire d'édition de l'article y seront et au final, le contrôleur ne fera qu'afficher cette vue sans savoir vraiment ce qu'il y a dedans. En pratique, c'est le designer d'un projet qui travaille sur les vues. Séparer vues et contrôleurs permet aux designers et développeurs PHP de travailler ensemble sans se marcher dessus.

Au final, si vous avez bien compris, le contrôleur ne contient que du code très simple, car il se contente d'utiliser des modèles et des vues en leur attribuant des tâches précises. Il agit un peu comme un chef d'orchestre, qui n'agit qu'une baguette alors que ses musiciens jouent des instruments complexes.

Parcours d'une requête dans Symfony2

Afin de bien visualiser tous les acteurs que nous avons vu jusqu'à présent, je vous propose un schéma du parcours complet d'une requête dans Symfony2 :



En le parcourant avec des mots, voici ce que cela donne :

1. Le visiteur demande la page /blog ;
2. Le contrôleur frontal reçoit la requête, charge le Kernel et la lui transmet ;
3. Le Kernel demande au Routeur quel contrôleur exécuter pour l'URL /blog. Ce Routeur est un composant Symfony2 qui fait la correspondance entre URL et contrôleurs, nous l'étudierons bien sûr dans un prochain chapitre. Le Routeur fait donc son travail, et dit au Kernel qu'il faut exécuter le contrôleur SdzBlog ;
4. Le Kernel exécute donc ce contrôleur. Le contrôleur demande au modèle Article la liste des articles, puis la donne à la vue ListeArticles pour qu'elle construise la page HTML et la lui retourne. Une fois fini, le contrôleur envoie au visiteur la page HTML complète.

J'ai mis des couleurs pour distinguer les points où on intervient. En vert, donc le contrôleur, modèle et vue, c'est ce qu'on devra développer nous-mêmes. En orange, donc le Kernel et le Routeur, c'est ce qu'on devra configurer. On ne touchera pas au contrôleur frontal, en gris.

Maintenant, il ne nous reste plus qu'à voir comment organiser concrètement notre code à nous et sa configuration.

Symfony2 et ses bundles

La découpe en « bundles »

Le concept

Vous avez déjà croisé ce terme de bundle quelquefois depuis le début du cours, mais qu'est-ce qui se cache derrière ce terme ?

Pour faire simple, un bundle est une brique de votre application. Symfony2 utilise ce concept novateur qui consiste à regrouper dans un même endroit, le bundle, tout ce qui concerne une même fonctionnalité. Par exemple, on peut imaginer un bundle Blog dans notre site, qui regrouperait les contrôleurs, les modèles, les vues, les fichiers CSS et JavaScript, etc. Tout ce qui concerne directement la fonctionnalité blog de notre site.

Cette organisation permet de découper naturellement nos fonctionnalités, et ainsi de ranger chaque fichier à sa place. Un fichier JavaScript n'est utilisé que sur le bundle blog ? Mettez-le dans le bundle blog ! Bien évidemment, au sein d'un bundle, il faut retrouver également une architecture bien définie, nous l'étudions juste après.

Des exemples

Pour mieux visualiser, je vous propose quelques bons exemples de bundles possibles :

- Un bundle Utilisateur, qui va gérer les utilisateurs ainsi que les groupes, intégrer des pages d'administration de ces utilisateurs, et des pages classiques comme le formulaire d'inscription, de récupération de mot de passe, etc.
- Un bundle Blog, qui va fournir une interface pour gérer un blog sur le site. Ce bundle peut utiliser le bundle Utilisateur pour faire un lien vers les profils des auteurs des articles et des commentaires.
- Un bundle Boutique, qui va fournir des outils pour gérer des produits et des commandes.
- Un bundle Admin, qui va fournir uniquement une interface vers les outils d'administration des différents bundles utilisés (utilisateur, blog, etc.). Attention il ne doit pas y avoir beaucoup de code dans ce bundle, ce n'est qu'un raccourci vers les fonctionnalités d'administration des autres bundles. La partie admin pour ajouter un article au blog *doit* se trouver dans le bundle blog.

Et ces bundles, parce qu'ils respectent des règles communes, vont fonctionner ensemble. Par exemple, un bundle Forum et un bundle Utilisateur devront s'entendre : dans un forum, ce sont des utilisateurs qui interagissent. 😊

L'intérêt

Une question à toujours se poser : quel est l'intérêt de ce que l'on est en train de faire ?

En plus d'organiser votre code par fonctionnalités, la découpe en bundles permet l'échange de bundles entre applications ! Cela signifie que vous pouvez développer une fonctionnalité, puis la partager avec d'autres développeurs ou encore la réutiliser dans un de vos autres projets. Et bien entendu, cela marche dans l'autre sens : vous pouvez installer dans votre projet des bundles qui ont été développés par d'autres !

Le principe même des bundles offre donc des possibilités infinies ! Imaginez le nombre de fonctionnalités classiques sur un site internet, que vous n'aurez plus à développer vous-mêmes. Vous avez besoin d'un livre d'or ? Il existe sûrement un bundle. Vous avez besoin d'un blog ? Il existe sûrement un bundle. Etc.

Les bundles de la communauté

Presque tous les bundles de la communauté Symfony2 sont regroupés sur un même site : <http://knxbundles.com/>. Il en existe beaucoup, et pour n'en citer que quelques uns :

- **FOSUserBundle** : c'est un *bundle* destiné à gérer les utilisateurs de votre site. Concrètement, il fournit le modèle « utilisateur » ainsi que le contrôleur pour accomplir les actions de base (connexion, inscription, déconnexion, édition d'un utilisateur, etc.) et fournit aussi les vues qui vont avec. Bref, il suffit d'installer le *bundle* et de le personnaliser un peu pour obtenir un espace membre !
- **FOSCommentBundle** : c'est un bundle destiné à gérer des commentaires. Concrètement, il fournit le modèle « commentaire » (ainsi que son contrôleur) pour ajouter, modifier et supprimer les commentaires. Les vues sont fournies avec, évidemment. Bref, en installant ce bundle, vous pourrez ajouter un fil de commentaires à n'importe quelle page de votre site !
- **GravatarBundle** : c'est un bundle destiné à gérer les avatars depuis le service web **Gravatar**. Concrètement, il fournit une extension au moteur de template pour pouvoir afficher facilement un avatar issu de Gravatar via une simple fonction qui s'avère être très pratique ;
- etc.

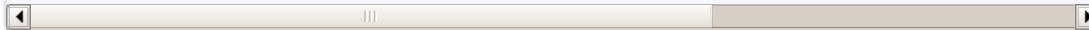
Je vous conseille vivement de passer sur <http://knxbundles.com/> avant de commencer à développer un bundle. S'il en existe déjà un et qu'il vous convient, il serait trop bête de réinventer la roue. 😊 Bien sûr, il faut d'abord apprendre à installer un bundle externe, patience !

La structure d'un bundle

Un bundle contient tout : contrôleurs, vues, modèles, classes personnelles, etc. Bref, tout ce qu'il faut pour remplir la fonction du bundle. Évidemment, tout cela est organisé en dossiers afin que tout le monde s'y retrouve. Voici la structure d'un bundle à partir de son répertoire de base, vous pouvez en voir l'illustration grâce au bundle exemple fourni par défaut dans `src/Acme/DemoBundle/` :

Code : Console

```
/Controller      | Contient vos contrôleurs  
/DependencyInjection | Contient des informations sur votre bundle (chargement autom  
/Entity          | Contient vos modèles  
/Form            | Contient vos éventuels formulaires  
  
/Resources  
-- /config       | Contient les fichiers de configuration de votre bundle (nc  
-- /public        | Contient les fichiers publics de votre bundle : fichiers C  
-- /views         | Contient les vues de notre bundle, les templates Twig  
/Tests           | Contient vos éventuels tests unitaires et fonctionnels. Nous
```



La structure est assez simple au final, retenez-la bien. Sachez qu'elle n'est pas du tout fixe, vous pouvez créer tous les dossiers que vous voulez pour mieux organiser votre code. Mais cette structure conventionnelle permet à d'autres développeurs de comprendre rapidement votre bundle. Bien entendu je vous guiderai pour chaque création de fichier. 😊

En résumé

- Symfony2 est organisé en 4 répertoires : app, src, vendor et web ;
- Le répertoire dans lequel on passera le plus de temps est src, il contient le code source de notre site ;
- Il existe deux environnements de travail :
 - L'environnement prod est destiné à vos visiteurs : il est rapide à exécuter, et ne divulgue pas les messages d'erreur ;
 - L'environnement dev est destiné au développeur, vous : il est plus lent mais offre plein d'informations utiles au développement.
- Symfony2 utilise l'architecture MVC pour bien organiser les différentes parties du code source ;
- Un bundle est une brique de votre application : il contient tout ce qui concerne une fonctionnalité donnée. Cela permet de bien organiser les différentes parties de votre site ;
- Il existe des milliers de bundles développés par la communauté, pensez à vérifier qu'il n'existe pas déjà un bundle qui fait ce que vous souhaitez faire !

Utilisons la console pour créer un bundle

Dans ce chapitre nous allons créer notre premier bundle, juste histoire d'avoir la structure de base de notre code futur. Mais nous ne le ferons pas n'importe comment : nous allons générer le bundle en utilisant une commande Symfony2 en console ! L'objectif est de découvrir la console utilement.

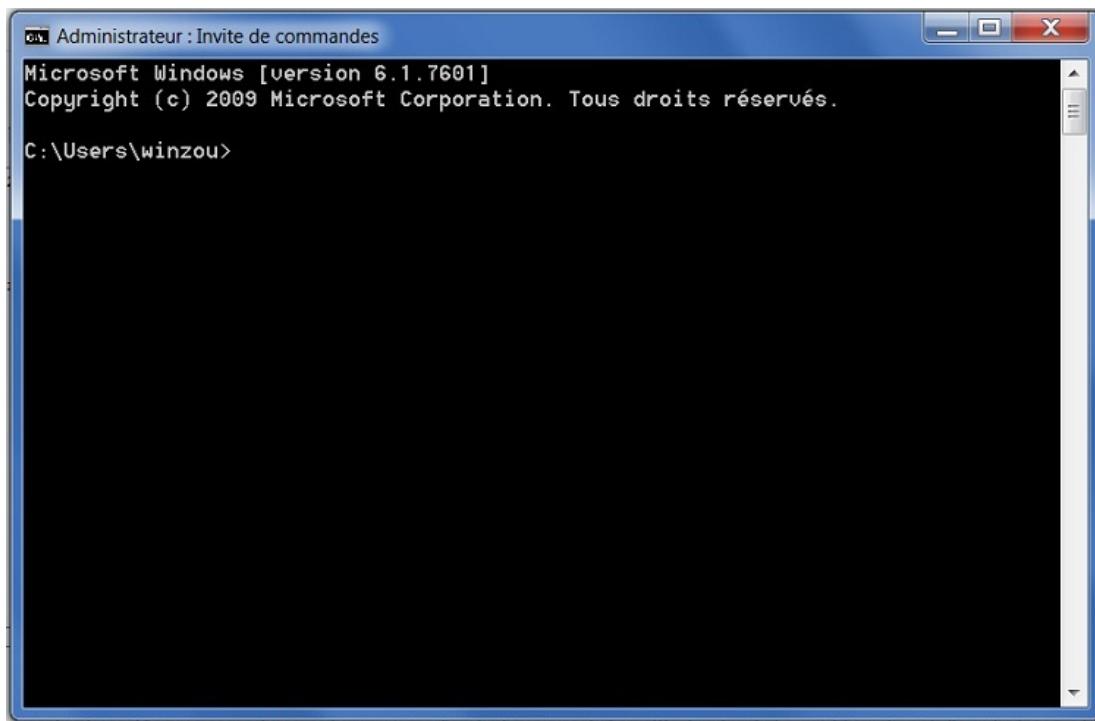
Utilisation de la console

Tout d'abord, vous devez savoir une chose : Symfony2 intègre des commandes disponibles non pas *via* le navigateur, mais *via* l'invite de commandes (sous Windows) ou le terminal (sous Linux). Il existe pas mal de commandes qui vont nous servir assez souvent lors du développement, apprenons donc dès maintenant à utiliser cette console !

Les outils disponibles en ligne de commandes ont pour objectif de nous faciliter la vie. Ce n'est pas un obscur programme pour les geeks amoureux de la console ! Vous pourrez à partir de là générer une base de code source pour certains fichiers récurrents, vider le cache, ajouter des utilisateurs par la suite, etc. N'ayez pas peur de cette console.

Sous Windows

Lancez l'invite de commandes : Menu Démarrer > Programmes > Accessoires > Invite de commandes. Une fenêtre semblable à la figure suivante devrait apparaître.



La console Windows

Puis placez-vous dans le répertoire où vous avez mis Symfony2, en utilisant la commande Windows `cd` (je vous laisse adapter la commande) :

Code : Console

```
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Users\winzou>cd ../../wamp/www/Symfony

C:\wamp\www\Symfony>_
```



Vous avez également la possibilité de vous rendre dans le répertoire où vous avez mis Symfony2 *via* l'explorateur de fichiers et de faire un clic droit sur le dossier Symfony2 en appuyant en même temps sur la touche Shift de votre clavier. Dans le menu contextuel, choisissez Ouvrir une fenêtre de commandes ici.

On va exécuter des fichiers PHP depuis cette invite de commandes. En l'occurrence c'est le fichier `app/console` (ouvrez-le, c'est bien du PHP) que nous allons exécuter. Pour cela, il faut lancer la commande PHP avec le nom du fichier en argument : `php app/console`. C'est parti :

Code : Console

```
C:\wamp\www\Symfony>php app/console

Symfony version 2.1.0-DEV - app/dev/debug
```

```

Usage:
[options] command [arguments]

Options:
--help           -h Display this help message.
--quiet          -q Do not output any message.
--verbose         -v Increase verbosity of messages.
--version         -V Display this application version.
--ansi            Force ANSI output.
--no-ansi          Disable ANSI output.
--no-interaction -n Do not ask any interactive question.
--shell            -s Launch the shell.
--process-
isolation      Launch commands from shell as a separate processes.
--env             -e The Environment name.
--no-debug        Switches off debug mode.

```

Et voilà, vous venez d'exécuter une commande Symfony ! Celle-ci ne fait pas grand chose, c'était juste un entraînement.



La commande ne fonctionne pas ? On vous dit que PHP n'est pas un exécutable ? Vous avez dû oublier d'ajouter PHP dans votre variable PATH, on l'a fait à la fin du [premier chapitre](#), jetez-y un oeil.

Sous Linux et Mac

Ouvrez le terminal. Placez-vous dans le répertoire où vous avez mis Symfony2, probablement `/var/www` pour Linux ou `/user/sites` pour Mac. Le fichier que nous allons exécuter est `app/console`, il faut donc lancer la commande `php app/console`. Je ne vous fais pas de capture d'écran, mais j'imagine que vous savez le faire !

À quoi ça sert ?

Une très bonne question, qu'il faut toujours se poser. 😊 La réponse est très simple : à nous simplifier la vie !

Depuis cette console, on pourra par exemple créer une base de données, vider le cache, ajouter ou modifier des utilisateurs (sans passer par phpMyAdmin !), etc. Mais ce qui nous intéresse dans ce chapitre, c'est la **génération de code**.

En effet, pour créer un bundle, un modèle ou un formulaire, le code de départ est toujours le même. C'est ce code là que le générateur va écrire pour nous. Du temps de gagné !

Comment ça marche ?



Comment Symfony2, un framework pourtant écrit en PHP, peut-il avoir des commandes en console ?

Vous devez savoir que PHP peut s'exécuter depuis le navigateur, mais également depuis la console. En fait côté Symfony2, tout est toujours écrit en PHP, il n'y a rien d'autre. Pour en être sûr, ouvrez le fichier `app/console` :

Code : PHP

```

<?php

require_once __DIR__.'/bootstrap.php.cache';
require_once __DIR__.'/AppKernel.php';

use Symfony\Bundle\FrameworkBundle\Console\Application;
use Symfony\Component\Console\Input\ArgvInput;

$input = new ArgvInput();
$env = $input->getParameterOption(array('--env', '-e'),
getenv('SYMFONY_ENV') ?: 'dev');
$debug = getenv('SYMFONY_DEBUG') !== '0' && !$input-
>hasParameterOption(array('--no-debug', '')) && $env !== 'prod';

$kernel = new AppKernel($env, $debug);
$application = new Application($kernel);
$application->run();

```

Vous ne remarquez rien ? Il ressemble beaucoup au contrôleur frontal `app.php` ! En fait il fait presque la même chose, il inclut les mêmes fichiers, et charge également le Kernel. Mais il définit la requête comme venant de la console, ce qui exécute du code différent par la suite. On pourra nous aussi écrire du code qui sera exécuté non pas depuis le navigateur (comme les contrôleurs habituels), mais depuis la console. Rien ne change pour le code, si ce n'est que l'affichage ne peut pas être en HTML bien

évidemment.

Créons notre bundle

Tout est bundle

Rappelez-vous : dans Symfony2, chaque partie de votre site est un bundle. Pour créer notre première page, il faut donc d'abord créer notre premier bundle. Rassurez-vous, créer un bundle est extrêmement simple avec le générateur. Démonstration !

Exécuter la bonne commande

Comme on vient de l'apprendre, exécutez la commande `php app/console generate:bundle`.

1. Choisir le namespace

Symfony2 vous demande le namespace de votre bundle :

Code : Console

```
C:\wamp\www\Symfony>php app/console generate:bundle

Welcome to the Symfony2 bundle generator

Your application code must be written in bundles. This command helps
you generate them easily.

Each bundle is hosted under a namespace (like Acme/Bundle/BlogBundle).
The namespace should begin with a "vendor" name like your company name, your
project name, or your client name, followed by one or more optional category
sub-namespaces, and it should end with the bundle name itself
(which must have Bundle as a suffix).

See http://symfony.com/doc/current/cookbook/bundles/best_practices.html#index-
1
for more
details on bundle naming conventions.

Use / instead of \ for the namespace delimiter to avoid any problem.

Bundle namespace:_
```

Vous pouvez nommer votre namespace comme bon vous semble, il faut juste qu'il se termine par le suffixe « Bundle ». Par convention, on le compose de trois parties. Nous allons nommer notre namespace « Sdz\BlogBundle ». Explications :

1. « Sdz » est le namespace racine : il vous représente. Vous pouvez mettre votre pseudo, le nom de votre site ou ce que vous voulez ;
2. « Blog » est le nom du bundle en lui-même : il définit ce que fait le bundle. Ici, nous créons un blog, nous l'avons donc simplement appelé « Blog » ;
3. « Bundle » est le suffixe obligatoire.

Rentrez donc dans la console `Sdz/BlogBundle`, avec des slashes juste pour cette fois pour les besoins de la console, mais un namespace comprend bien des anti-slashes.

2. Choisir le nom

Symfony2 vous demande le nom de votre bundle :

Code : Console

```
Bundle namespace: Sdz/BlogBundle

In your code, a bundle is often referenced by its name. It can be the
concatenation of all namespace parts but it's really up to you to come
up with a unique name (a good practice is to start with the vendor name).
Based on the namespace, we suggest SdzBlogBundle.

Bundle name [SdzBlogBundle]:_
```

Par convention, on nomme le bundle de la même manière que le namespace, sans les slashes. On a donc : SdzBlogBundle. C'est ce que Symfony2 vous propose par défaut (la valeur entre les crochets), appuyez donc simplement sur Entrée. Retenez ce nom, par la suite quand on parlera du nom du bundle, cela voudra dire ce nom là : SdzBlogBundle.

3. Choisir la destination

Symfony2 vous demande l'endroit où vous voulez que les fichiers du bundle soient générés :

Code : Console

```
The bundle can be generated anywhere. The suggested default directory uses  
the standard conventions.
```

```
Target directory [C:/wamp/www/src]:_
```

Par convention, comme on l'a vu, on place nos bundles dans le répertoire `src/`. C'est ce que Symfony2 vous propose, appuyez donc sur Entrée.

4. Choisir le format de configuration

Symfony2 vous demande sous quelle forme vous voulez configurer votre bundle. Il s'agit simplement du format de la configuration, que nous ferons plus tard. Il existe plusieurs moyens comme vous pouvez le voir : Yaml, XML, PHP ou Annotations.

Code : Console

```
Target directory [C:/wamp/www/src]:
```

```
Determine the format to use for the generated configuration.
```

```
Configuration format (yml, xml, php, or annotation) [annotation]:
```

Chacune a ses avantages et inconvénients. Nous allons utiliser le Yaml (yml) ici, car c'est bien adapté pour un bundle. Mais sachez que nous utiliserons les annotations pour nos futures entités par exemple. Entrez donc `yml`.

5. Choisir quelle structure générer

Symfony2 vous demande si vous voulez générer juste le minimum ou une structure plus complète pour le bundle :

Code : Console

```
Configuration format (yml, xml, php, or annotation) [annotation]: yml
```

```
To help you get started faster, the command can generate some  
code snippets for you.
```

```
Do you want to generate the whole directory structure [no]?
```

Faisons simple et demandons à Symfony2 de tout nous générer. Entrez donc `yes`.

6. Confirmez, et c'est joué !

Pour toutes les questions suivantes, confirmez en appuyant sur Entrée à chaque fois. Et voilà, votre bundle est généré :

Code : Console

```
Do you want to generate the whole directory structure [no]? yes
```

```
Summary before generation
```

```
You are going to generate a "Sdz\BlogBundle\SdzBlogBundle" bundle  
in "C:/wamp/www/Symfony/src/" using the "yml" format.
```

```
Do you confirm generation [yes]?
```

```
Bundle generation
```

```
Generating the bundle code: OK  
Checking that the bundle is autoloaded: OK  
Confirm automatic update of your Kernel [yes]?  
Enabling the bundle inside the Kernel: OK  
Confirm automatic update of the Routing [yes]?  
Importing the bundle routing resource: OK
```

You can now start using the generated code!

C:\wamp\www\Symfony_



Tout d'abord, je vous réserve une petite surprise : allez voir sur http://localhost/Symfony/web/app_dev.php/hello/winzou ! Le bundle est déjà opérationnel ! Le code-exemple généré affiche le texte passé dans l'URL, vous pouvez donc également essayer ceci : http://localhost/Symfony/web/app_dev.php/hello/World.



Mais, pourquoi il n'y a pas la *toolbar* en bas de la page ?

C'est normal, c'est juste un petit truc à savoir pour éviter de s'arracher les cheveux inutilement 😱. La *toolbar* est un petit bout de code HTML que rajoute Symfony2 à chaque page... contenant la balise </body>. Or sur cette page, vous pouvez afficher la source depuis votre navigateur, il n'y a aucune balise HTML en fait, donc Symfony2 n'ajoute pas la *toolbar*.

Pour l'activer rien de plus simple, il nous faut rajouter une toute petite structure HTML. Pour cela ouvrez le fichier src/Sdz/BlogBundle/Resources/views/Default/index.html.twig, c'est la vue utilisée pour cette page. L'extension .twig signifie qu'on utilise le moteur de templates Twig pour gérer nos vues, on en reparlera bien sûr. Le fichier est plutôt simple, et je vous propose de le changer ainsi :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Default/index.html.twig #-}

<html>
  <body>
    Hello {{ name }} !
  </body>
</html>
```

Actualisez la page, et voilà une magnifique *toolbar* semblable à la figure suivante qui apparaît en bas de la page ! Seule la balise </body> suffisait, mais quitte à changer autant avoir une structure HTML valide. 😊



Que s'est-il passé ?

Dans les coulisses, Symfony2 a fait pas mal de choses, revoyons tout ça à notre rythme.

Symfony2 a généré la structure du bundle

Allez dans le répertoire src/Sdz/BlogBundle, vous pouvez voir tout ce que Symfony2 a généré pour nous. Rappelez-vous la structure d'un bundle que nous avons vu au chapitre précédent : Symfony2 en a généré la majorité !

À savoir : le seul fichier obligatoire pour un bundle est en fait la classe SdzBlogBundle.php à la racine du répertoire. Vous pouvez l'ouvrir et voir ce qu'il contient : pas très intéressant en soi ; heureusement que Symfony l'a généré tout seul. Sachez-le dès maintenant : nous ne modifierons presque jamais ce fichier, vous pouvez passer votre chemin.

Symfony2 a enregistré notre bundle auprès du Kernel

Le bundle est créé, mais il faut dire à Symfony2 de le charger. Pour cela il faut configurer le noyau (le Kernel) pour qu'il le charge. Rappelez-vous, la configuration de l'application se trouve dans le répertoire /app. En l'occurrence, la configuration du noyau se fait dans le fichier app/AppKernel.php :

Code : PHP

```
<?php

use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
```

```

{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
        new
    Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new JMS\AopBundle\JMSAopBundle(),
        new JMS\DiExtraBundle\JMSSDIExtraBundle($this),
        new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
        new Sdz\BlogBundle\SdzBlogBundle(), // Le générateur a rajouté
    cette ligne
    );

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

    return $bundles;
}

// ...
}

```

Cette classe permet donc uniquement de définir quels bundles charger pour l'application. Vous pouvez le voir, ils sont instanciés dans un simple tableau. Les lignes 11 à 22 définissent les bundles à charger pour l'environnement de production. Les lignes 26 à 29 définissent les bundles à charger *en plus* pour l'environnement de développement.

Comme vous pouvez le voir, le générateur du bundle a modifié lui-même ce fichier pour y rajouter la ligne 22. C'est ce que l'on appelle « enregistrer le bundle dans l'application ».

Vous pouvez voir également qu'il en existe plein d'autres, ce sont tous les bundles par défaut qui apportent des fonctionnalités de base au framework Symfony2. En fait, quand on parle de Symfony2, on parle à la fois de ses composants (Kernel, Routeur, etc.) et de ses bundles.

Symfony2 a enregistré nos routes auprès du Routeur



Les routes ? Le Routeur ?

Pas de panique nous verrons tout cela dans les prochains chapitres. Sachez juste pour l'instant que le rôle du Routeur, que nous avons brièvement vu sur le schéma du chapitre précédent, est de déterminer quel contrôleur exécuter en fonction de l'URL appelée. Pour cela, il utilise les routes.

Chaque bundle dispose de ses propres routes. Pour notre bundle fraîchement créé, vous pouvez les voir dans le fichier `src/Sdz/Bundle/Resources/config/routing.yml`. En l'occurrence il n'y en a qu'une seule :

Code : YAML

```

# src/Sdz/Bundle/Resources/config/routing.yml

SdzBlogBundle_homepage:
    pattern: /hello/{name}
    defaults: { _controller: SdzBlogBundle:Default:index }

```

Or ces routes ne sont pas chargées automatiquement, il faut dire au Routeur « *Bonjour, mon bundle SdzBlogBundle contient des routes qu'il faut que tu viennes chercher* ». Cela se fait, vous l'aurez deviné, dans la configuration de l'application. Cette configuration se trouve toujours dans le répertoire `app/`, en l'occurrence pour les routes il s'agit du fichier `app/config/routing.yml` :

Code : YAML

```

# app/config/routing.yml

SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix: /

```

Ce sont ces lignes qui importent le fichier de routes situé dans notre bundle. Ces lignes ont déjà été générées par le générateur de bundle, vraiment pratique lui !

À retenir

Ce qu'il faut retenir de tout cela, c'est que pour qu'un bundle soit opérationnel il faut :

- Son code source, situé dans `src/Application/Bundle`, et dont le seul fichier obligatoire est la classe à la racine `SdzBlogBundle.php` ;
- Enregistrer le bundle dans le noyau pour qu'il soit chargé, en modifiant le fichier `app/AppKernel.php` ;
- Enregistrer les routes (si le bundle en contient) dans le Routeur pour qu'elles soient chargées, en modifiant le fichier `app/config/routing.yml`.

Ces trois points sont bien sûr effectués automatiquement lorsqu'on utilise le générateur. Mais vous pouvez tout à fait créer un bundle sans l'utiliser, et il faudra alors remplir cette petite *checklist*.

Par la suite, tout notre code source sera situé dans des bundles. Un moyen très propre de bien structurer son application.

En résumé

- Les commandes Symfony2 disponibles en ligne de commandes ont pour objectif de nous faciliter la vie en automatisant certaines tâches ;
- Les commandes sont faites, comme tout Symfony2, en PHP uniquement. La console n'est qu'un moyen différent du navigateur pour exécuter du code PHP ;
- La commande pour générer un nouveau bundle est `php app/console generate:bundle`.

Partie 2 : Les bases de Symfony2 : « Hello World »

Cette partie a pour objectif de créer une première page, et d'en maîtriser les composantes. Ce sont ici les notions les plus importantes de Symfony2, prenez donc votre temps pour bien comprendre tout ce qui est abordé.

Mon premier Hello World avec Symfony2

L'objectif de ce chapitre est de créer de toute pièce notre première page avec Symfony2 : une simple page blanche comprenant un « Hello World! ». Nous allons donc créer tous les éléments indispensables pour concevoir une telle page.

Nous allons avoir une vue d'ensemble de tous les acteurs qui interviennent dans la création d'une page : routeur, contrôleur et template. Pour cela, tous les détails ne seront pas expliqués afin de se concentrer sur l'essentiel : la façon dont ils se coordonnent. Vous devrez attendre les prochains chapitres pour étudier un à un ces trois acteurs, patience donc !

Ne bloquez donc pas sur un point si vous ne comprenez pas tout, forcez-vous juste à comprendre l'ensemble. À la fin du chapitre, vous aurez une vision globale de la création d'une page et l'objectif sera atteint.

Bonne lecture !

Créons notre route

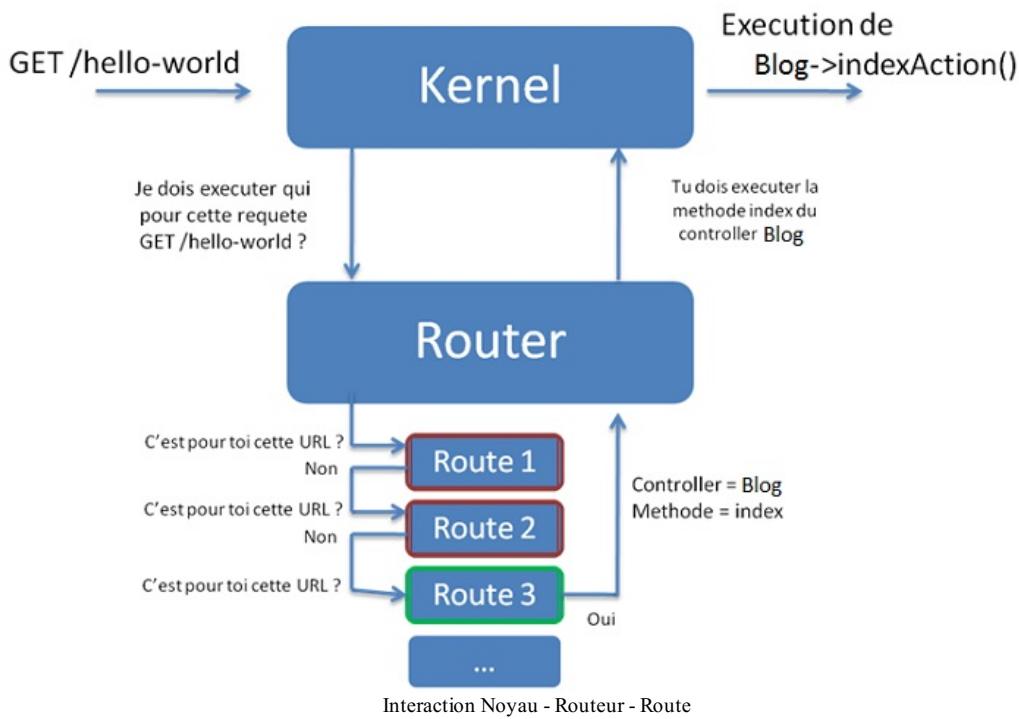
Nous travaillons dans notre bundle SdzBlogBundle, placez vous donc dans son répertoire : `src/Sdz/BlogBundle`.

Pour créer une page, il faut d'abord définir l'URL à laquelle elle sera accessible. Pour cela, il faut créer la route de cette page.

Le routeur (ou « router ») ? Une route ?

Objectif

L'objectif du routeur est de dire à Symfony2 ce qu'il doit faire lorsque l'on appelle l'URL `/hello-world` (par exemple). Nous devons donc créer une route qui va dire : « lorsque l'on est sur l'URL `/hello-world`, alors on appelle le contrôleur « Blog » qui va afficher un Hello World. ». Regardez la figure suivante.



Comme je l'ai dit, nous ne toucherons ni au noyau, ni au routeur : nous nous occuperons juste des routes.

1. Créons notre fichier de routes

Les routes se définissent dans un simple fichier texte, que Symfony2 a déjà généré pour notre SdzBlogBundle. Usuellement, on nomme ce fichier `Resources/config/routing.yml` dans le répertoire du bundle. Ouvrez le fichier, et rajoutez cette route à la suite de celle qui existe déjà :

Code : YAML

```

# src/Sdz/BlogBundle/Resources/config/routing.yml

HelloTheWorld:
    pattern:  /hello-world
    defaults: { _controller: SdzBlogBundle:Blog:index }
  
```

Vous venez de créer votre première route !



Contrairement à ce que vous pourriez penser, l'indentation se fait avec 4 espaces par niveau, *pas avec des tabulations* ! Je le précise parce qu'un jour vous ferez l'erreur (l'inévitable ne peut être évité), et vous me remercieriez de vous avoir mis sur la voie. Et ceci est valable pour tous vos fichiers .yml.



Attention également, il semble y avoir des erreurs lors des copier-coller depuis le tutoriel vers les fichiers .yml. Si vous rencontrez une obscure erreur, pensez à bien définir l'encodage du fichier en « UTF-8 sans BOM » et à supprimer les éventuels caractères non désirés. C'est un bogue étrange qui provient du site, mais dont on ne connaît pas l'origine. L'esquive est de toujours recopier les exemples YAML que je vous donne, et de ne pas les copier-coller.

Essayons de comprendre rapidement cette route :

- HelloTheWorld est le nom de la route. Il est assez arbitraire, et vous permet juste de vous y retrouver par la suite. La seule contrainte est qu'il soit unique ;
- pattern correspond à l'URL à laquelle nous souhaitons que notre *hello world* soit accessible. C'est ce qui permet à la route de dire : « *cette URL est pour moi, je prends* » ;
- defaults correspond aux paramètres de la routes, dont :
 - _controller, qui correspond à l'**action** (ici, « index ») que l'on veut exécuter et au **contrôleur** (ici, « Blog ») que l'on va appeler (un contrôleur peut contenir plusieurs actions, c'est à dire plusieurs pages).

Ne vous inquiétez pas, un chapitre complet est dédié au routeur et vous permettra de jouer avec. Pour l'instant ce fichier nous permet juste d'avancer.

Mais avant d'aller plus loin, penchons-nous sur la valeur que l'on a donnée à `_controller` : « `SdzBlogBundle:Blog:index` ». Cette valeur se découpe en suivant les deux-points (« : ») :

- « `SdzBlogBundle` » est le nom de notre bundle, celui dans lequel Symfony2 ira chercher le contrôleur.
- « `Blog` » est le nom du contrôleur à ouvrir. En terme de fichier, cela correspond à `controller/BlogController.php` dans le répertoire du bundle. Dans notre cas, nous avons `src/Sdz/BlogBundle/controller/BlogController.php` comme chemin absolu.
- « `index` » est le nom de la méthode à exécuter au sein du contrôleur.

2. Informons Symfony2 que nous avons des routes pour lui

On l'a vu précédemment, grâce au bon travail du générateur, Symfony2 est déjà au courant du fichier de routes de notre bundle. Mais ce n'est pas par magie ! Il faut que vous sachiez comment tout cela s'imbrique. Ouvrez le fichier de configuration globale de notre application : `app/config/config.yml`. Dans ce fichier, il y a plein de valeurs, mais la section qui nous intéresse est la section « router », à la ligne 9 que je vous remets ici :

Code : YAML

```
# app/config/config.yml

router:
    resource: "%kernel.root_dir%/config/routing.yml"
    strict_parameters: %kernel.debug%
```

Cette section indique au routeur qu'il doit chercher les routes dans le fichier `app/config/routing.yml` (%`kernel.root_dir`% est un paramètre qui vaut « `app` » dans notre cas). Le routeur va donc se contenter d'ouvrir ce fichier. Ouvrez-le également :

Code : YAML

```
# app/config/routing.yml

SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix:   /
```

Outre les commentaires, vous voyez que le générateur a inséré une route spéciale (qui n'a pas de « pattern », mais une « resource ») qui va importer le fichier de routes de notre bundle.



C'est parce que ce fichier `routing.yml` était vide (avant la génération du bundle) que l'on avait une erreur « `page introuvable` » en « `prod` » : comme il n'y a aucune route définie, Symfony2 nous informe à juste titre qu'aucune page n'existe. Et si le mode « `dev` » ne nous donnait pas d'erreur, c'est parce qu'il charge le fichier `routing_dev.yml` et non `routing.yml`. Et dans ce fichier, allez voir, il y a bien quelques routes définies. Et il y a aussi la ligne qui importe le fichier `routing.yml`, afin d'avoir les routes du mode « `prod` » dans le mode « `dev` » (l'inverse étant bien sûr faux).

Bref, vous n'avez rien à modifier ici, c'était juste pour que vous sachiez que l'import du fichier de routes d'un bundle n'est pas automatique, il se définit dans le fichier de routes global.

Revenons à nos moutons. En fait, on aurait pu ajouter notre route `HelloTheWorld` directement dans ce fichier `routing.yml`. Cela aurait fonctionné et cela aurait été plutôt rapide. Mais c'est oublier notre découpage en bundles ! En effet, cette route concerne le bundle du blog, elle doit donc se trouver dans notre bundle et pas ailleurs. N'oubliez jamais ce principe.

Cela permet à notre bundle d'être indépendant : si plus tard nous ajoutons, modifions ou supprimons des routes dans notre bundle, nous ne toucherons qu'au fichier `src/Sdz/BlogBundle/Resources/config/routing.yml` au lieu de `app/config/routing.yml`. 😊

Et voilà, il n'y a plus qu'à créer le fameux contrôleur Blog ainsi que sa méthode `index` !

Créons notre contrôleur

Le rôle du contrôleur

Rappelez-vous ce que nous avons dit sur le MVC :

- le contrôleur est la « glu » de notre site ;
- il « utilise » tous les autres composants (base de données, formulaires, templates, etc.) pour générer la réponse suite à notre requête ;
- c'est ici que résidera toute la logique de notre site : si l'utilisateur est connecté et qu'il a le droit de modifier cet article, alors j'affiche le formulaire d'édition des articles de mon blog.

Créons notre contrôleur

1. Le fichier de notre contrôleur Blog

Dans un bundle, les contrôleurs se trouvent dans le répertoire `Controller` du bundle. Rappelez-vous : dans la route, on a dit qu'il fallait faire appel au contrôleur nommé « Blog ». Le nom des fichiers des contrôleurs doit respecter une convention très simple : il doit commencer par le nom du contrôleur, ici « Blog », suivi du suffixe « `Controller` ». Au final, on doit donc créer le fichier `src/Sdz/BlogBundle/Controller/BlogController.php`.

Même si Symfony2 a déjà créé un contrôleur `DefaultController` pour nous, ce n'est qu'un exemple, on va utiliser le nôtre. Ouvrez donc notre `BlogController.php` et mettez-y le code suivant :

Code : PHP

```
<?php  
// src/Sdz/BlogBundle/Controller/BlogController.php  
  
namespace Sdz\BlogBundle\Controller;  
  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\HttpFoundation\Response;  
  
class BlogController extends Controller  
{  
    public function indexAction()  
    {  
        return new Response("Hello World !");  
    }  
}
```



Surprise : allez voir sur http://localhost/Symfony/web/app_dev.php/hello-world ! Même bundle mais contrôleur différent, on en fait des choses !

Maintenant, essayons de comprendre rapidement ce fichier :

- ligne 5 : on se place dans le namespace des contrôleurs de notre bundle. Rien de bien compliqué, suivez la structure des répertoires dans lequel se trouve le contrôleur ;
- ligne 7 : notre contrôleur hérite de ce contrôleur de base. Il faut donc le charger grâce au `use` ;
- ligne 8 : notre contrôleur va utiliser l'objet `Response`, il faut donc le charger grâce au `use` ;
- ligne 10 : le nom de notre contrôleur respecte le nom du fichier pour que l'autoload fonctionne ;
- ligne 12 : on définit la méthode `indexAction()`. N'oubliez pas de mettre le suffixe `Action` derrière le nom de la méthode ;
- ligne 14 : on crée une réponse toute simple. L'argument de l'objet `Response` est le contenu de la page que vous envoyez au visiteur, ici « Hello World ! ». Puis on retourne cet objet.

Bon : certes, le rendu n'est pas très joli, mais au moins, nous avons atteint l'objectif d'afficher nous-mêmes un Hello World.



Pourquoi `indexAction()` ? J'ai pas suivi là.

En effet il faut savoir que le nom des méthodes des contrôleur doit respecter une convention. Lorsque dans la route on parle de l'action « index », dans le contrôleur on doit définir la méthode `indexAction()`, c'est-à-dire le nom de l'action suivi du suffixe « Action », tout simplement. Il n'y a pas tellement à réfléchir, c'est une simple convention pour distinguer les méthodes qui vont être appelées par le noyau (les `xxxAction()`) des autres méthodes que vous pourriez créer au sein de votre contrôleur.

Mais écrire le contenu de sa page de cette manière dans le contrôleur, ça n'est pas très pratique, et en plus de cela, on ne respecte pas le modèle MVC. Utilisons donc les templates !

Créons notre template Twig

Les templates avec Twig

Savez-vous ce qu'est un moteur de template ? C'est un script qui permet d'utiliser des templates, c'est-à-dire des fichiers qui ont pour but d'afficher le contenu de votre page HTML de façon dynamique, mais sans PHP. Comment ? Avec leur langage à eux. Chaque moteur a son propre langage.

Avec Symfony2, nous allons employer le moteur Twig. Voici un exemple de comparaison entre un template simple en PHP (premier code) et un template en « langage Twig » (deuxième code).

Code : PHP

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bienvenue dans Symfony2 !</title>
  </head>
  <body>
    <h1><?php echo $titre_page; ?></h1>

    <ul id="navigation">
      <?php foreach ($navigation as $item) { ?>
        <li>
          <a href="<?php echo $item->getHref(); ?>"><?php echo
$item->getTitle(); ?></a>
        </li>
      <?php } ?>
    </ul>
  </body>
</html>
```

Code : HTML & Django

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bienvenue dans Symfony2 !</title>
  </head>
  <body>
    <h1>{{ titre_page }}</h1>

    <ul id="navigation">
      {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.titre }}</a></li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Ils se ressemblent, soyons d'accord. Mais celui réalisé avec Twig est bien plus facile à lire ! Pour afficher une variable, vous faites juste `{{ ma_var }}` au lieu de `<?php echo $ma_var; ?>`.

Le but en fait est de faciliter le travail de votre designer. Un designer ne connaît pas forcément le PHP, ni forcément Twig d'ailleurs. Mais Twig est très rapide à prendre en main, plus rapide à écrire et à lire, et il dispose aussi de fonctionnalités très intéressantes. Par exemple, imaginons que votre designer veuille mettre les titres en lettres majuscules (COMME CECI). Il lui suffit de faire : `{{ titre|upper }}`, où `titre` est la variable qui contient le titre d'un article de blog par exemple. C'est plus joli que `<?php echo strtoupper($titre); ?>` non ?

Nous verrons dans le chapitre dédié à Twig les nombreuses fonctionnalités que le moteur vous propose et qui vont vous faciliter la vie. En attendant, nous devons avancer sur notre « Hello World ».

Utiliser Twig avec Symfony2

Comment utiliser un template Twig depuis notre contrôleur, au lieu d'afficher notre texte tout simple ?

1. Créons le fichier du template

Le répertoire des templates (ou vues) d'un bundle se trouve dans le dossier `Resources/views`. Ici encore on ne va pas

utiliser le template situé dans le répertoire Default généré par Symfony2. Créons notre propre répertoire Blog et créons notre template index.html.twig dans ce répertoire. Nous avons donc le fichier src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig.

Je vous propose de découper ce nom Blog/index.html.twig :

- Blog/ est le nom du répertoire. Nous l'avons appelé comme notre contrôleur afin de nous y retrouver (ce n'est pas une obligation, mais c'est fortement recommandé) ;
- index est le nom de notre template qui est aussi le nom de la méthode de notre contrôleur (*idem*, pas obligatoire, mais recommandé) ;
- html correspond au format du contenu de notre template. Ici, nous allons y mettre du code HTML, mais vous serez amené à vouloir y mettre du XML ou autre : vous changerez donc cette extension. Cela permet de mieux s'y retrouver ;
- twig est le format de notre template. Ici, nous utilisons Twig comme moteur de template, mais il est toujours possible d'utiliser des templates PHP.

Revenez à notre template et mettez ce code à l'intérieur :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #-}

<!DOCTYPE html>
<html>
  <head>
    <title>Bienvenue sur ma première page avec le Site du Zéro
  </title>
  </head>
  <body>
    <h1>Hello World !</h1>

    <p>
      Le Hello World est un grand classique en programmation.
      Il signifie énormément, car cela veut dire que vous avez
      réussi à exécuter le programme pour accomplir une tâche simple
    :
      afficher ce hello world !
    </p>
  </body>
</html>
```

Dans ce template, nous n'avons utilisé ni variable, ni structure Twig. En fait, c'est un simple fichier contenant du code HTML pur !

2. Appelons ce template depuis le contrôleur

Il ne reste plus qu'à appeler ce template. C'est le rôle du contrôleur, c'est donc au sein de la méthode `indexAction()` que nous allons appeler le template. Cela se fait très simplement avec la méthode `$this->render()`. Cette méthode prend en paramètre le nom du template et retourne un objet de type `Response` avec pour contenu le contenu de notre template. Voici le contrôleur modifié en conséquence :

Code : PHP

```
<?php

// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function indexAction()
    {
        return $this->render('SdzBlogBundle:Blog:index.html.twig');
    }
}
```

Nous n'avons modifié que la ligne 14. La convention pour le nom du template est la même que pour le nom du contrôleur, souvenez-vous : NomDuBundle:NomDuContrôleur:NomDeLAction.

Maintenant, retournez sur la page http://localhost/Symfony/web/app_dev.php/hello-world et profitez !



Vous avez des problèmes d'accent ? Faites attention à bien définir l'encodage de vos templates en UTF8 Sans BOM.



Notez également l'apparition de la *toolbar* en bas de la page. Je vous l'avais déjà dit, Symfony2 l'ajoute automatiquement lorsqu'il détecte la balise fermante `</body>`. C'est pour cela que nous ne l'avions pas tout à l'heure avec notre « Hello World » tout simple.

Vous voulez vous amuser un peu avec les variables Twig ? Modifiez la ligne 14 du contrôleur pour rajouter un 2^e argument à la méthode `render()` :

Code : PHP

```
<?php
return $this->render('SdzBlogBundle:Blog:index.html.twig',
array('nom' => 'winzou'));
```

Puis modifiez votre template en remplaçant la balise h1 par la suivante :

Code : HTML

```
<h1>Hello {{ nom }} !</h1>
```

C'est tout ! Rechargez la page. Bonjour à vous également. 😊 On verra plus en détails le passage de variables dans le chapitre dédié à Twig bien évidemment.

Notre objectif : créer un blog

Le fil conducteur : un blog

Tout au long de ce cours, nous construirons un blog. Cela me permet d'utiliser des exemples cohérents entre eux et de vous montrer comment construire un blog de toutes pièces. Bien sûr, libre à vous d'adapter les exemples au projet que vous souhaitez mener, je vous y encourage, même !

Le choix du blog n'est pas très original, mais il permet que l'on se comprenne bien : vous savez déjà ce qu'est un blog, vous comprendrez donc, en théorie, tous les exemples.

Notre blog

Le blog que nous allons créer est très simple. En voici les grandes lignes :

- nous aurons des articles auxquels nous attacherons des tags ;
- nous pourrons lire, écrire, éditer et rechercher des articles ;
- nous pourrons créer, modifier et supprimer des tags ;
- au début, nous n'aurons pas de système de gestion des utilisateurs : nous devrons saisir notre nom lorsque nous rédigerais un article. Puis nous rajouterons la couche utilisateur ;
- au début, il n'y aura pas de système de commentaires. Puis nous ajouterons cette couche commentaire.

Un peu de nettoyage

Avec tous les éléments générés par Symfony2 et les nôtres, il y a un peu de redondance. Vous pouvez donc supprimer joyeusement :

- Le contrôleur `Controller/DefaultController.php` ;
- Les vues dans le répertoire `Resources/views/Default` ;
- La route `SdzBlogBundle_homepage` dans `Resources/config/routing.yml`.

Ainsi que tout ce qui concerne le bundle `AcmeDemoBundle`, un bundle de démonstration intégré dans la distribution standard de Symfony2 et dont nous ne nous servirons pas. Supprimez donc :

- Le répertoire `src/Acme` ;
- La ligne 26 du fichier `app/AppKernel.php`, celle qui active le bundle `AcmeDemoBundle` (`$bundles[] = new Acme\DemoBundle\AcmeDemoBundle();`) ;
- Les 3 premières routes dans le fichier `app/config/routing_dev.yml` (`_welcome`, `_demo_secured` et `_demo`).

Schéma de développement sous Symfony2

Si vous rafraîchissez la page pour vérifier que tout est bon, il est possible que vous ayez une erreur ! En effet il faut prendre dès maintenant un réflexe Symfony2 : vider le cache. Car Symfony, pour nous offrir autant de fonctionnalités et être si rapide, utilise beaucoup son cache (des calculs qu'il ne fait qu'une fois puis qu'il stocke). Or après certaines modifications, le cache n'est plus à jour et il se peut que cela génère des erreurs. Deux cas de figure :

- En mode prod, c'est simple, Symfony2 ne vide jamais le cache. Ça lui permet de ne faire aucune vérification sur la validité du cache (ce qui prend du temps), et de servir les pages très rapidement à vos visiteurs. La solution : vider le cache à la main *à chaque fois que vous faites des changements*. Cela se fait grâce à la commande `php app/console cache:clear --env=prod`.
- En mode dev, c'est plus simple et plus compliqué. Lorsque vous modifiez votre code, Symfony reconstruit une bonne partie du cache à la prochaine page que vous chargez. Donc pas forcément besoin de vider le cache. Seulement, comme il ne reconstruit pas tout, il peut parfois apparaître des bugs. Dans ce cas, un petit `php app/console cache:clear` résout le problème en 3 secondes !



Parfois, il se peut que la commande `cache:clear` génère des erreurs lors de son exécution. Dans ce cas, essayez de relancer la commande, parfois une deuxième passe peut résoudre les problèmes. Dans le cas contraire, supprimez le cache à la main en supprimant simplement le répertoire `app/cache/dev` (ou `app/cache/prod` suivant l'environnement).

Typiquement, un schéma classique de développement est le suivant :

- Je fais des changements, je teste ;
- Je fais des changements, je teste ;
- Je fais des changements, je teste ça ne marche pas, je vide le cache ça marche ;
- Je fais des changements, je teste ;
- Je fais des changements, je teste ;
- Je fais des changements, je teste ça ne marche pas, je vide le cache ça marche ;
- ...
- En fin de journée, j'envoie tout sur le serveur de production, je vide obligatoirement le cache, je teste, ça marche.

Évidemment, quand je dis « je teste ça ne marche pas », j'entends « ça devrait marcher et l'erreur rencontrée est étrange ». Si vous faites une erreur dans votre propre code, ce n'est pas un `cache:clear` qui va la résoudre !

Pour conclure

Et voilà, nous avons créé une page de A à Z ! Voici plusieurs remarques sur ce chapitre.

D'abord, ne vous affolez pas si vous n'avez pas tout compris. Le but de ce chapitre était de vous donner une vision globale d'une page Symfony2. Vous avez des notions de *bundles*, de routes, de contrôleurs et de *templates* : vous savez presque tout ! Il ne reste plus qu'à approfondir chacune de ces notions, ce que nous ferons dès le prochain chapitre.

Ensuite, sachez que *tout* n'est pas à refaire lorsque vous créez une deuxième page. Je vous invite là, maintenant, à créer une page `/byebye-world` et voyez si vous y arrivez. Dans le cas contraire relisez ce chapitre, puis si vous ne trouvez pas votre erreur, n'hésitez pas à poser votre question sur le [forum PHP](#), d'autres Zéros qui sont passés par là seront ravis de vous aider.



Sur le forum, pensez à mettre le tag [Symfony2] dans le titre de votre sujet, afin de s'y retrouver.

Enfin, préparez-vous pour la suite, les choses sérieuses commencent !

En résumé

- Le rôle du routeur est de déterminer quel route utiliser pour la requête courante ;
- Le rôle d'une route est d'associer une URL à une action de contrôleur ;
- Le rôle du contrôleur est de retourner au noyau un objet Response, qui contient la réponse HTTP à envoyer à l'internaute (page HTML ou redirection) ;
- Le rôle des vues est de mettre en forme les données que le contrôleur lui donne, afin de former une page HTML, un flux RSS, un email, etc.

Le routeur de Symfony2

Comme nous avons pu l'apercevoir, le rôle du routeur est, à partir d'une URL, de déterminer quel contrôleur appeler et avec quels arguments. Cela permet de configurer son application pour avoir de très belles URL, ce qui est important pour le référencement et même pour le confort des visiteurs. Soyons d'accord, l'URL `/article/le-système-de-route` est bien plus sexy que `index.php?contrôleur=article&méthode=voir&id=5` !

Vous avez sans doute déjà entendu parler d'*URL Rewriting* ? Le routeur, bien que différent, permet effectivement de faire l'équivalent de l'*URL Rewriting*, mais il le fait côté PHP, et donc est bien mieux intégré à notre code.

Le fonctionnement

L'objectif de ce chapitre est de vous transmettre toutes les connaissances pour pouvoir créer ce que l'on appelle un fichier de *mapping* des routes (un fichier de « correspondances » en français). Ce fichier, généralement situé dans `votreBundle/Resources/config/routing.yml`, contient la définition des routes. Chaque route fait la correspondance entre une URL et le contrôleur à appeler. Je vous invite à mettre dès maintenant les routes présentées au code suivant dans le fichier, nous allons travailler dessus dans ce chapitre :

Code : YAML

```
# src/sdz/BlogBundle/Resources/config/routing.yml

sdzblog_accueil:
    pattern:    /blog
    defaults:   { _controller: SdzBlogBundle:Blog:index }

sdzblog_voir:
    pattern:    /blog/article/{id}
    defaults:   { _controller: SdzBlogBundle:Blog:voir }

sdzblog_ajouter:
    pattern:    /blog/ajouter
    defaults:   { _controller: SdzBlogBundle:Blog:ajouter }
```



Petit rappel au cas où : l'indentation se fait avec 4 espaces par niveau, et non avec des tabulations.



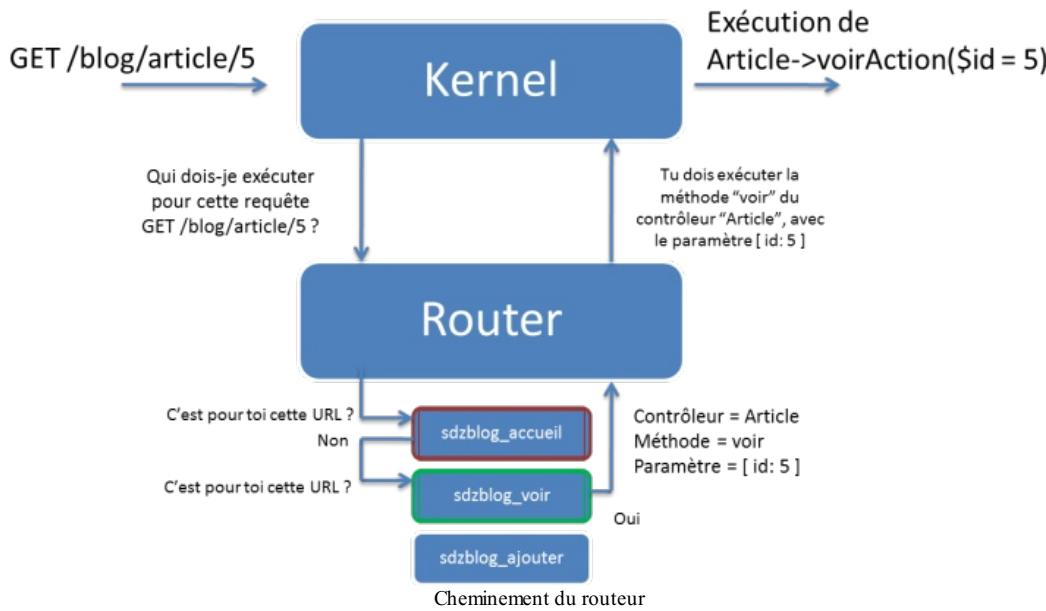
Vous pouvez supprimer la route `helloTheWorld` que nous avons créée au chapitre précédent, elle ne nous resservira plus. Vous pouvez voir qu'à la place nous avons maintenant une route `sdzblog_accueil`, qui pointe vers la même action du contrôleur. 😊

Fonctionnement du routeur

Dans le code précédent, vous pouvez distinguer trois blocs. Chacun correspond à une route. Nous les verrons en détail plus loin, mais vous pouvez constater que chaque route prend :

- une entrée (ligne `pattern`) : c'est l'URL à capturer ;
- une sortie (ligne `defaults`) : ce sont les paramètres de la route, notamment celui qui dit quel est le contrôleur à appeler.

Le but du routeur est donc, à partir d'une URL, de trouver la route correspondante et de retourner le contrôleur que veut cette route. Pour trouver la bonne route, le routeur va les parcourir une par une, dans l'ordre du fichier, et s'arrêter à la première route qui fonctionne. La figure suivante est un schéma équivalent au chapitre précédent, mais actualisée pour notre fichier de route précédent.



Et voici en texte le fonctionnement, pas à pas :

1. On appelle l'URL `/blog/article/5`;
2. Le routeur essaie de faire correspondre cette URL avec le pattern de la première route. Ici, `/blog/article/5` ne correspond pas du tout à `/blog` (ligne pattern de la première route);
3. Le routeur passe donc à la route suivante. Il essaie de faire correspondre `/blog/article/5` avec `/blog/article/{id}`. Nous le verrons plus loin, mais `{id}` est un paramètre, une sorte de joker « je prends tout ». Cette route correspond, car nous avons bien :
 - o `/blog/article` (URL) = `/blog/article` (route)
 - o `5` (URL) = `{id}` (route);
4. Le routeur s'arrête donc, il a trouvé sa route ;
5. Il demande à la route : « *Quel contrôleur souhaites-tu appeler, et avec quels paramètres ?* », la route répond « *Je veux le contrôleur `SdzBlogBundle:Blog:voir`, avec le paramètre `$id = 5`.* »;
6. Le routeur renvoie donc ces informations au kernel (le noyau de Symfony2) ;
7. Le noyau va exécuter le bon contrôleur !

Dans le cas où le routeur ne trouve aucune route correspondante, le noyau de Symfony2 va déclencher une erreur 404.

Pour chaque page, il est possible de visualiser toutes les routes que le routeur essaie une à une, et celle qu'il utilise finalement. C'est le **Profiler** qui s'occupe de tracer cela, accessible depuis la barre d'outil : cliquez sur le nom de la route dans la barre d'outils, « `sdzblog_accueil` » si vous êtes sur la page `/blog`. Ce lien vous amène dans l'onglet « *Request* » du **Profiler**, mais allez dans l'onglet « *Routing* » qui nous intéresse. Vous devriez obtenir la figure suivante.

View all Profile for: GET http://localhost/Symfony/web/app_dev.php/blog by 127.0.0.1 at Wed, 11 Jul 2012 22:13:52 +0200

Routing for "/blog"

Route: `sdzblog_accueil`

Route parameters: No parameters

Route matching logs

Route name	Pattern	Log
<code>_wdt</code>	<code>/_wdt/{token}</code>	Pattern " <code>/_wdt/{token}</code> " does not match
<code>_profiler_search</code>	<code>/_profiler/search</code>	Pattern " <code>/_profiler/search</code> " does not match
<code>_profiler_purge</code>	<code>/_profiler/purge</code>	Pattern " <code>/_profiler/purge</code> " does not match
<code>_profiler_info</code>	<code>/_profiler/info/{about}</code>	Pattern " <code>/_profiler/info/{about}</code> " does not match
<code>_profiler_import</code>	<code>/_profiler/import</code>	Pattern " <code>/_profiler/import</code> " does not match
<code>_profiler_export</code>	<code>/_profiler/export/{token}.txt</code>	Pattern " <code>/_profiler/export/{token}.txt</code> " does not match
<code>_profiler_phpinfo</code>	<code>/_profiler/phpinfo</code>	Pattern " <code>/_profiler/phpinfo</code> " does not match
<code>_profiler_search_results</code>	<code>/_profiler/{token}/search/results</code>	Pattern " <code>/_profiler/{token}/search/results</code> " does not match
<code>_profiler</code>	<code>/_profiler/{token}</code>	Pattern " <code>/_profiler/{token}</code> " does not match
<code>_profiler_redirect</code>	<code>/_profiler/</code>	Pattern " <code>/_profiler/</code> " does not match
<code>_configurator_home</code>	<code>/_configurator/</code>	Pattern " <code>/_configurator/</code> " does not match
<code>_configurator_step</code>	<code>/_configurator/step/{index}</code>	Pattern " <code>/_configurator/step/{index}</code> " does not match
<code>_configurator_final</code>	<code>/_configurator/final</code>	Pattern " <code>/_configurator/final</code> " does not match
<code>sdzblog_accueil</code>	<code>/blog</code>	Route matches!

Liste des routes enregistrées par le routeur



Vous pouvez voir qu'il y a déjà pas mal de routes définies alors que nous n'avons rien fait. Ces routes qui commencent par `/_profiler/` sont les routes nécessaires au Profiler, dans lequel vous êtes. Eh oui, c'est un bundle également, le bundle WebProfilerBundle !

Convention pour le nom du contrôleur

Vous l'avez vu, lorsque l'on définit le contrôleur à appeler dans la route, il y a une convention à respecter : la même que pour appeler un template (nous l'avons vu au chapitre précédent). Un rappel ne fait pas de mal : lorsque vous écrivez « `SdzBlogBundle:Blog:voir` », vous avez trois informations :

- « `SdzBlogBundle` » est le nom du bundle dans lequel aller chercher le contrôleur. En terme de fichier, cela signifie pour Symfony2 : « *Voir dans le répertoire de ce bundle.* ». Dans notre cas, Symfony2 ira voir dans `src/Sdz/BlogBundle` ;
- « `Blog` » est le nom du contrôleur à ouvrir. En terme de fichier, cela correspond à `controller/BlogController.php` dans le répertoire du bundle. Dans notre cas, nous avons comme chemin absolu `src/Sdz/BlogBundle/controller/BlogController.php` ;
- « `voir` » est le nom de l'action à exécuter au sein du contrôleur. Attention lorsque vous définissez cette méthode dans le contrôleur, vous devez la faire suivre du suffixe « `Action` », comme ceci : `<?php public function voirAction()`.

Les routes de base

Créer une route

Étudions la première route plus en détail :

Code : YAML

```
# src/Sdz/BlogBundle/config/Resources/routing.yml

sdzblog_accueil:
    pattern: /blog
    defaults: { _controller: SdzBlogBundle:Blog:index }
```

Ce bloc représente ce que l'on nomme une « route ». Elle est constituée au minimum de trois éléments :

- `sdzblog_accueil` est le nom de la route. Il n'a aucune importance dans le travail du routeur, mais il interviendra lorsque l'on voudra générer des URL : eh oui, on n'écrira pas l'URL à la main mais on fera appel au routeur pour qu'il fasse le travail à notre place ! Retenez donc pour l'instant qu'il faut qu'un nom soit unique et clair. On a donc préfixé les routes de « `sdzblog` » pour l'unicité entre bundles ;
- `pattern: /blog` est l'URL sur laquelle la route s'applique. Ici, « `/blog` » correspond à une URL absolue du type `http://www.monsite.com/blog` ;
- `defaults: { _controller: SdzBlogBundle:Blog:index }` correspond au contrôleur à appeler.

Vous avez maintenant les bases pour créer une route simple !

Créer une route avec des paramètres

Reprendons la deuxième route de notre exemple :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_voir:
    pattern: /blog/article/{id}
    defaults: { _controller: SdzBlogBundle:Blog:voir }
```

Grâce au paramètre `{id}` dans le pattern de notre route, toutes les URL du type `/blog/article/*` seront gérées par cette route, par exemple : `/blog/article/5` ou `/blog/article/654`, ou même `/blog/article/sodfihsodfih` (on n'a pas encore dit que `{id}` devait être un nombre, patience !). Par contre, l'URL `/blog/article` ne sera pas interceptée car le paramètre `{id}` n'est pas renseigné. En effet, les paramètres sont par défaut obligatoires, nous verrons quand et comment les rendre facultatifs plus loin dans ce chapitre.

Mais si le routeur s'arrêtait là, il n'aurait aucun intérêt. Toute sa puissance réside dans le fait que ce paramètre `{id}` est accessible depuis votre contrôleur ! Si vous appelez l'URL `/blog/article/5`, alors depuis votre contrôleur, vous aurez la variable `$id` (du nom du paramètre) qui aura pour valeur « `5` ». Je vous invite à créer la méthode correspondante dans le contrôleur :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php
```

```

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    // ... ici la méthode indexAction() que l'on a déjà créée

    // La route fait appel à SdzBlogBundle:Blog:voir, on doit donc
    // définir la méthode voirAction
    // On donne à cette méthode l'argument $id, pour correspondre au
    // paramètre {id} de la route
    public function voirAction($id)
    {
        // $id vaut 5 si l'on a appellé l'URL /blog/article/5

        // Ici, on récupèrera depuis la base de données l'article
        // correspondant à l'id $id
        // Puis on passera l'article à la vue pour qu'elle puisse
        // l'afficher

        return new Response("Affichage de l'article d'id : ".$id.".");
    }
}

```

N'oubliez pas de tester votre code à l'adresse suivante : http://localhost/Symfony/web/app_dev.php/blog/article/5, et amusez-vous à changer la valeur du paramètre.

Vous pouvez bien sûr multiplier les paramètres au sein d'une même route. Rajoutez cette route juste après la route « `sdzblog_voir` », pour l'exemple :

Code : YAML

```

# src/Sdz/Bundle/Resources/config/routing.yml

sdzblog_voir_slug:
    pattern:    /blog/{annee}/{slug}.{format}
    defaults:   { _controller: SdzBlogBundle:Blog:voirSlug }

```

Cette route permet d'intercepter les URL suivantes : `/blog/2011/mon-weekend.html` ou `/blog/2012/symfony.xml`, etc. Et voici la méthode correspondante qu'on aurait côté contrôleur :

Code : PHP

```

<?php
// src/Sdz/Bundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    // ... ici les méthodes indexAction() et voirAction() que l'on a
    // déjà créées

    // On récupère tous les paramètres en argument de la méthode
    public function voirSlugAction($slug, $annee, $format)
    {
        // Ici le contenu de la méthode
        return new Response("On pourrait afficher l'article
        correspondant au slug '".$slug."', créé en ".$annee." et au format
        ".$format.".");
    }
}

```



Notez que l'ordre des arguments dans la définition de la méthode `voirSlugAction()` n'a pas d'importance. La route fait la correspondance à partir du nom des variables utilisées, non à partir de leur ordre. C'est toujours bon à savoir !

Revenez à notre route et notez également le point entre les paramètres `{slug}` et `{format}` : vous pouvez en effet séparer vos paramètres soit avec le slash (`« / »`), soit avec le point (`« . »`). Veillez donc à ne pas utiliser de point dans le contenu de vos paramètres. Par exemple, pour notre paramètre `{slug}`, une URL `/blog/2011/mon-weekend.était.bien.html` ne va pas correspondre à cette route car :

- {annee} = 2011 ;
- {slug} = mon-weekend ;
- {format} = etait ;
- ? = bien ;
- ? = html ;

La route attend des paramètres à mettre en face de ces dernières valeurs, et comme il n'y en a pas, cette route dit : « *Cette URL ne me correspond pas, passez à la route suivante* ». Attention donc à ce petit détail. 😊

Les routes avancées

Créer une route avec des paramètres et leurs contraintes

Nous avons créé une route avec des paramètres, très bien. Mais si quelqu'un essaie d'atteindre l'URL /blog/oaisd/aouish.oasidh, eh bien rien ne l'en empêche ! Et pourtant, « oaisd » n'est pas tellement une année valide ! La solution ? Les contraintes sur les paramètres. Reprenons notre dernière route sdzbog_voir_slug :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
sdzbog_voir_slug:
    pattern:    /blog/{annee}/{slug}.{format}
    defaults:  { _controller: SdzBlogBundle:Blog:voirSlug }
```

Nous voulons ne récupérer que les bonnes URL où l'année vaut « 2010 » et non « oshidf », par exemple. Cette dernière devrait retourner une erreur 404 (page introuvable). Pour cela, il nous suffit qu'aucune route ne l'intercepte ; ainsi, le routeur arrivera à la fin du fichier sans aucune route correspondante et il déclenchera tout seul une erreur 404.

Comment faire pour que notre paramètre {annee} n'intercepte pas « oshidf » ? C'est très simple :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
sdzbog_voir_slug:
    pattern:    /blog/{annee}/{slug}.{format}
    defaults:  { _controller: SdzBlogBundle:Blog:voirSlug }
    requirements:
        annee: \d{4}
        format: html|xml
```

Nous avons ajouté la section `requirements`. Comme vous pouvez le voir, on utilise les expressions régulières pour déterminer les contraintes que doivent respecter les paramètres. Ici :

- \d{4} veut dire « *quatre chiffres à la suite* ». L'URL /blog/sdff/mon-weekend.html ne sera donc pas interceptée. Vous l'avez reconnu, c'est une expression régulière. Vous pouvez utiliser n'importe laquelle, je vous invite à lire [le cours correspondant de M@teo](#) ;
- html|xml signifie « *soit html, soit xml* ». L'URL /blog/2011/mon-weekend.rss ne sera donc pas interceptée.



N'hésitez surtout pas à faire les tests ! Cette route est opérationnelle, nous avons créé l'action correspondante dans le contrôleur. Essayez donc de bien comprendre quels paramètres sont valides, lesquels ne le sont pas. Vous pouvez également changer la section `requirements`.

Maintenant, nous souhaitons aller plus loin. En effet, si le « .xml » est utile pour récupérer l'article au format XML (pourquoi pas ?), le « .html » semble inutile : par défaut, le visiteur veut toujours du HTML. Il faut donc rendre le paramètre {format} facultatif.

Utiliser des paramètres facultatifs

Reprendons notre route et ajoutons-y la possibilité à {format} de ne pas être renseigné :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
sdzbog_voir_slug:
    pattern:    /blog/{annee}/{slug}.{format}
    defaults:  { _controller: SdzBlogBundle:Blog:voirSlug, format: html }
    requirements:
        annee: \d{4}
        format: html|xml
```

Nous avons juste rajouté une valeur par défaut dans le tableau `defaults : format: html`. C'est aussi simple que cela ! Ainsi, l'URL `/blog/2011/mon-weekend` sera bien interceptée et le paramètre `format` sera mis à sa valeur par défaut, à savoir « `html` ». Au niveau du contrôleur, rien ne change : vous gardez l'argument `$format` comme avant et celui-ci vaudra « `html` », la valeur par défaut.

Utiliser des « paramètres système »

Prenons l'exemple de notre paramètre `{format}` : lorsqu'il vaut « `xml` », vous allez afficher du XML et devrez donc envoyer le header avec le bon Content-type. Les développeurs de Symfony2 ont pensé à nous et prévu des « paramètres système ». Ils s'utilisent exactement comme des paramètres classiques, mais effectuent automatiquement des actions supplémentaires :

- le paramètre `{_format}` : lorsqu'il est utilisé (comme notre paramètre `{format}`, rajoutez juste un *underscore*), alors un header avec le Content-type correspondant est envoyé. Exemple : vous appelez `/blog/2011/mon-weekend.xml` et le routeur va dire à l'objet Request que l'utilisateur demande du XML. Ainsi, l'objet Response enverra un header `Content-type: application/xml`. Vous n'avez plus à vous en soucier ! Depuis le contrôleur, vous pouvez récupérer ce format soit avec l'argument `$_format` comme n'importe quel autre argument, soit via la méthode `getRequestFormat()` de l'objet Request. Par exemple : `<?php $this->get('request')->getRequestFormat();`
- le paramètre `{_locale}` : lorsqu'il est utilisé, il va définir la langue dans laquelle l'utilisateur souhaite obtenir la page. Ainsi, si vous avez défini des fichiers de traduction ou si vous employez des bundles qui en utilisent, alors les traductions dans la langue du paramètre `{_locale}` seront chargées. Pensez à mettre un `requirements` sur la valeur de ce paramètre pour éviter que vos utilisateurs ne demandent le russe alors que votre site n'est que bilingue français-anglais.

Ajouter un préfixe lors de l'import de nos routes

Vous avez remarqué que nous avons mis `/blog` au début du pattern de chacune de nos routes. En effet, on crée un blog, on aimerait donc que toutes les URL aient ce préfixe `/blog`. Au lieu de les répéter à chaque fois, Symfony2 vous propose de rajouter un préfixe lors de l'import du fichier de notre bundle.

Modifiez donc le fichier `app/config/routing.yml` comme suit :

Code : Autre

```
# app/config/routing.yml
SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix: /blog
```

Vous pouvez ainsi enlever la partie `/blog` de chacune de vos routes. Bonus : si un jour vous souhaitez changer `/blog` par `/blogdemichel`, vous n'aurez qu'à modifier une seule ligne. 😊

Générer des URL

Pourquoi générer des URL ?

J'ai mentionné précédemment que le routeur pouvait aussi générer des URL à partir du nom des routes. Ce n'est pas une fonctionnalité annexe, mais bien un outil puissant que nous avons là !

Par exemple, nous avons une route nommée « `sdzblog_voir` » qui écoute sur l'URL `/blog/article/{id}`. Vous décidez un jour de raccourcir vos URL et vous aimeriez bien que vos articles soient disponibles depuis `/blog/a/{id}`. Si vous aviez écrit toutes vos URL à la main, vous auriez dû toutes les changer à la main, une par une. Grâce à la génération d'URL, vous ne modifiez que la route : ainsi, toutes les URL générées seront mises à jour ! C'est un exemple simple, mais vous pouvez trouver des cas bien plus réels et tout aussi gênants sans la génération d'URL.

Comment générer des URL ?

1. Depuis le contrôleur

Pour générer une URL, vous devez le demander au routeur en lui donnant deux arguments : le nom de la route ainsi que les éventuels paramètres de cette route.

Depuis un contrôleur, c'est la méthode `<?php $this->generateUrl()` qu'il faut appeler. Par exemple :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php
namespace Sdz\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller;
```

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

class BlogController extends Controller
{
    public function indexAction()
    {
        // On fixe un id au hasard ici, il sera dynamique par la suite,
        // évidemment
        $id = 5;

        // On veut avoir l'URL de l'article d'id $id.
        $url = $this->generateUrl('sdzblog_voir', array('id' => $id));
        // $url vaut « /blog/article/5 »

        // On redirige vers cette URL (ça ne sert à rien, on est
        // d'accord, c'est pour l'exemple !)
        return $this->redirect($url);
    }
}
```

Pour générer une URL absolue, lorsque vous l'envoyez par e-mail, par exemple, il faut mettre le 3^e argument à true. Exemple :

Code : PHP

```
<?php
$url = $this->generateUrl('sdzblog_voir', array('id' => $id), true);
```

Ainsi, \$url vaut `http://monsite.com/blog/article/5` et pas uniquement `/blog/article/5`.

2. Depuis une vue Twig

Vous aurez bien plus l'occasion de devoir générer une URL depuis la vue. C'est la fonction `path` qu'il faut utiliser depuis un template Twig :

Code : HTML& Django

```
{# Dans une vue Twig, en considérant bien sûr que la variable
article_id est disponible #-}

<a href="{{ path('sdzblog_voir', { 'id': article_id }) }}>Lien vers
l'article d'id {{ article_id }}</a>
```

Et pour générer une URL absolue depuis Twig, pas de 3^e argument, mais on utilise la fonction `url()` au lieu de `path()`. Elle s'utilise exactement de la même manière, seul le nom change.

Voilà : vous savez générer des URL, ce n'était vraiment pas compliqué. Pensez bien à utiliser la fonction `{{ path }}` pour tous vos liens dans vos templates. 😊

Application : les routes de notre blog

Construction des routes

Revenons à notre blog. Maintenant que nous savons créer des routes, je vous propose de faire un premier jet de ce que seront nos URL. Voici les routes que je vous propose de créer, libre à vous d'en changer.

Page d'accueil

On souhaite avoir une URL très simple pour la page d'accueil : `/blog`. Comme `/blog` est défini comme préfixe lors du chargement des routes de notre bundle, le pattern ici est « `/` ». Mais on veut aussi pouvoir parcourir les articles plus anciens, donc il nous faut une notion de page courante. En rajoutant le paramètre facultatif `{page}`, nous aurons :

<code>/blog</code>	<code>page = 1</code>
<code>/blog/1</code>	<code>page = 1</code>
<code>/blog/2</code>	<code>page = 2</code>

C'est plutôt joli, non ? Voici la route :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml
```

```

sdzblog_accueil:
    pattern:    /{page}
    defaults:  { _controller: SdzBlogBundle:Blog:index, page: 1 }
    requirements:
        page: \d*

```

Page de visualisation d'un article

Pour la page d'un unique article, la route est très simple. Il suffit juste de bien mettre un paramètre {id} qui nous servira à récupérer le bon article côté contrôleur. Voici la route :

Code : YAML

```

# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_voir:
    pattern:    /article/{id}
    defaults:  { _controller: SdzBlogBundle:Blog:voir }

```

Ajout, modification et suppression

Les routes sont simples :

Code : YAML

```

# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_ajouter:
    pattern:    /ajouter
    defaults:  { _controller: SdzBlogBundle:Blog:ajouter }

sdzblog_modifier:
    pattern:    /modifier/{id}
    defaults:  { _controller: SdzBlogBundle:Blog:modifier }
    requirements:
        id: \d+

sdzblog_supprimer:
    pattern:    /supprimer/{id}
    defaults:  { _controller: SdzBlogBundle:Blog:supprimer }
    requirements:
        id: \d+

```

Récapitulatif

Voici le code complet de notre fichier `src/Sdz/BlogBundle/Resources/config/routing.yml`:

Code : YAML

```

# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_accueil:
    pattern:    /{page}
    defaults:  { _controller: SdzBlogBundle:Blog:index, page: 1 }
    requirements:
        page: \d*

sdzblog_voir:
    pattern:    /article/{id}
    defaults:  { _controller: SdzBlogBundle:Blog:voir }
    requirements:
        id: \d+

sdzblog_ajouter:
    pattern:    /ajouter
    defaults:  { _controller: SdzBlogBundle:Blog:ajouter }

sdzblog_modifier:
    pattern:    /modifier/{id}
    defaults:  { _controller: SdzBlogBundle:Blog:modifier }
    requirements:
        id: \d+

sdzblog_supprimer:

```

```
pattern: /supprimer/{id}
defaults: { _controller: SdzBlogBundle:Blog:supprimer }
requirements:
    id: \d+
```

N'oubliez pas de bien ajouter le préfixe `/blog` lors de l'import de ce fichier, dans `app/config/routing.yml`:

Code : YAML

```
# app/config/routing.yml

SdzBlogBundle:
    resource: "@SdzBlogBundle/Resources/config/routing.yml"
    prefix: /blog
```

Pour conclure

Ce chapitre est terminé, et vous savez maintenant tout ce qu'il faut savoir sur le routeur et les routes.

Retenez que ce système de route vous permet premièrement d'avoir des belles URL, et deuxièmement de découpler le nom de vos URL du nom de vos contrôleurs. Rajoutez à cela la génération d'URL, et vous avez un système extrêmement flexible et maintenable.

Le tout sans trop d'efforts !

Pour plus d'informations sur le système de route, n'hésitez pas à lire la [documentation officielle](#).

En résumé

- Une route est composée à minima de deux éléments : l'URL à faire correspondre (son *pattern*), et le contrôleur à exécuter (attribut *_controller*) ;
- Le routeur essaie de faire correspondre chaque route à l'URL appelée par l'internaute, et ce dans l'ordre d'apparition des routes : la première route qui correspond est sélectionnée ;
- Une route peut contenir des arguments, facultatifs ou non, représentés par les accolades `{argument}`, et dont leur valeur peut être soumises à des contraintes via la section *requirements* ;
- Le routeur est également capable de générer des URL à partir du nom d'une route, et de ses paramètres éventuels.

Les contrôleurs avec Symfony2

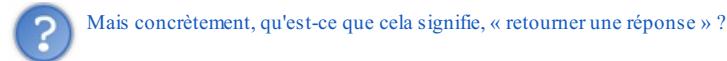
Ah, le contrôleur ! Vous le savez, c'est lui qui contient toute la logique de notre site Internet. Cependant, cela ne veut pas dire qu'il contient beaucoup de code. En fait, il ne fait qu'utiliser des services, les modèles et appeler la vue. Finalement, c'est un chef d'orchestre qui se contente de faire la liaison entre tout le monde.

Nous verrons dans ce chapitre ses droits, mais aussi son devoir le plus ultime : retourner une réponse !

Le rôle du contrôleur

Retourner une réponse

Je vous l'ai dit de nombreuses fois depuis le début de ce cours : le rôle du contrôleur est de retourner une réponse.



Mais concrètement, qu'est-ce que cela signifie, « retourner une réponse » ?

Souvenez-vous, Symfony2 s'est inspiré des concepts du protocole HTTP. Il existe dans Symfony2 une classe Response. Retourner une réponse signifie donc tout simplement : instancier un objet Response, disons \$response et faire un return \$response.

Voici le contrôleur le plus simple qui soit, c'est le contrôleur qu'on avait à créer dans un des chapitres précédents. Il dispose d'une seule méthode, nommée « index », et retourne une réponse qui ne contient que : « Hello World ! » :

Code : PHP

```
<?php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function indexAction()
    {
        return new Response("Hello World !");
    }
}
```

Et voilà, votre contrôleur remplit parfaitement son rôle !

Bien sûr, vous n'irez pas très loin en sachant juste cela. C'est pourquoi la suite de ce chapitre est découpée en 2 parties :

- les objets Request et Response qui vont vous permettre de construire une réponse en fonction de la requête ;
- les services qui vont vous permettre de réaliser tout le travail nécessaire pour préparer le contenu de votre réponse.

Manipuler l'objet Request

Les paramètres de la requête

Heureusement, toutes les requêtes que l'on peut faire sur un site Internet ne sont pas aussi simples que notre « Hello World ». Dans bien des cas, une requête contient des paramètres : l'id d'un article à afficher, le nom d'un membre à chercher dans la base de données, etc. Les paramètres sont la base de toute requête : la construction de la page à afficher dépend de chacun des paramètres en entrée.

Ces paramètres, nous savons déjà les gérer, nous l'avons vu dans le chapitre sur le routeur. Mais voici un petit rappel.

Les paramètres contenus dans les routes

Tout d'abord côté route, souvenez-vous on utilisait déjà des paramètres. Prenons l'exemple de la route sdzblog_voir :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_voir:
    pattern: /article/{id}
    defaults: { _controller: SdzBlogBundle:Blog:voir }
    requirements:
        id: \d+
```

Ici, le paramètre {id} de la requête est récupéré par la route, qui va le transformer en argument \$id pour le contrôleur. On a déjà fait la méthode correspondante dans le contrôleur, la voici pour rappel :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    // ...

    public function voirAction($id)
    {
        return new Response("Affichage de l'article d'id : ".$id."");
    }
}
```

Voici donc la première manière de récupérer des arguments : ceux contenus dans la route.

Les paramètres hors routes

En plus des paramètres de routes que nous venons de voir, vous pouvez récupérer les autres paramètres de l'URL, disons, « à l'ancienne ». Prenons par exemple l'URL /blog/article/5?tag=vacances, il nous faut bien un moyen pour récupérer ce paramètre tag ! C'est ici qu'intervient l'objet Request. Tout d'abord, voici comment récupérer la requête depuis un contrôleur :

Code : PHP

```
<?php
$request = $this->get('request');
```

Voilà, c'est aussi simple que ça ! La méthode get () du contrôleur, nous en reparlerons, permet d'accéder à toute sorte de services, dont notre requête. Maintenant que nous avons notre requête, récupérons nos paramètres :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    // ...

    public function voirAction($id)
    {
        // On récupère la requête
        $request = $this->get('request');

        // On récupère notre paramètre tag
        $tag = $request->query->get('tag');

        return new Response("Affichage de l'article d'id : ".$id.", avec
le tag : ".$tag);
    }
}
```

Et vous n'avez plus qu'à tester le résultat : /blog/article/9?tag=vacances.

Nous avons utilisé <?php \$request->query pour récupérer les paramètres de l'URL passés en GET, mais vous savez qu'il existe d'autres types de paramètres :

Type de paramètres	Méthode Symfony2	Méthode traditionnelle	Exemple
Variables d'URL	<?php \$request->query	\$_GET	<?php \$request->query->get('tag')
Variables de formulaire	<?php \$request->request	\$_POST	<?php \$request->request->get('tag')
Variables de cookie	<?php \$request->cookies	\$_COOKIE	<?php \$request->cookies->get('tag')

cookies	<code>>cookies</code>	<code>\$_COOKIE</code>	<code>>get('tag')</code>
Variables de serveur	<code><?php \$request->server</code>	<code>\$_SERVER</code>	<code><?php \$request->server->get('REQUEST_URI')</code>
Variables d'en-tête	<code><?php \$request->headers</code>	<code>\$_SERVER[HTTP_*]</code>	<code><?php \$request->headers->get('USER_AGENT')</code>
Paramètres de route	<code><?php \$request->attributes</code>	n/a	<code><?php \$request->attributes->get('id')</code> Est équivalent à : <code><?php \$id</code>

Avec cette façon d'accéder aux paramètres, vous n'avez pas besoin de tester leur existence. Par exemple, si vous faites `$request->query->get('sdf')` alors que le paramètre `sdf` n'est pas défini dans l'URL, cela vous retournera une chaîne vide, et non une erreur.

Les autres méthodes de l'objet Request

Heureusement, l'objet `Request` ne se limite pas à la récupération de paramètres. Il permet de savoir plusieurs choses intéressantes à propos de la requête en cours, voyons ses possibilités.

Récupérer la méthode de la requête HTTP

Pour savoir si la page a été récupérée via GET (clic sur un lien) ou via POST (envoi d'un formulaire), il existe la méthode `<?php $request->getMethod() ?>`:

Code : PHP

```
<?php
if( $request->getMethod() == 'POST' )
{
    // Un formulaire a été envoyé, on peut le traiter ici
}
```

Savoir si la requête est une requête Ajax

Lorsque vous utiliserez Ajax dans votre site, vous aurez sans doute besoin de savoir, depuis le contrôleur, si la requête en cours est une requête Ajax ou non. Par exemple, pour renvoyer du XML ou du JSON à la place du HTML. Pour cela, rien de plus simple !

Code : PHP

```
<?php
if( $request->isXmlHttpRequest() )
{
    // C'est une requête Ajax, retournons du JSON, par exemple
}
```

Toutes les autres

Pour avoir la liste exhaustive des méthodes disponibles sur l'objet `Request`, je vous invite à lire l'API de cet objet sur [le site de Symfony2](#). Vous y trouverez toutes les méthodes, même si nous en avons déjà survolé les principales.

Manipuler l'objet Response

Décomposition de la construction d'un objet Response

Pour que vous compreniez ce qu'il se passe en coulisse lors de la création d'une Réponse, voyons la manière longue et décomposée de construire et de retourner une réponse. Pour l'exemple, traitons le cas d'une page d'erreur 404 (page introuvable) :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    // On modifie voirAction car elle existe déjà
```

```

public function voirAction($id)
{
    // On crée la réponse sans lui donner de contenu pour le moment
    $response = new Response;

    // On définit le contenu.
    $response->setContent('Ceci est une page d\'erreur 404');

    // On définit le code HTTP
    // Rappelez-vous, 404 correspond à « page introuvable »
    $response->setStatusCode(404);

    // On retourne la réponse
    return $response;
}
}

```

N'hésitez pas à tester cette page, l'URL est http://localhost/Symfony/web/app_dev.php/blog/article/5 si vous avez gardé les mêmes routes depuis le début.

Je ne vous le cache pas : nous n'utiliserons jamais cette longue méthode ! Lisez plutôt la suite.

Réponses et vues

Généralement, vous préférerez que votre réponse soit contenue dans une vue, dans un template. Heureusement pour nous, le contrôleur dispose d'un raccourci : la méthode `<?php $this->render()`. Elle prend en paramètre le nom du template et ses variables, puis s'occupe de tout : créer la réponse, y passer le contenu du template, et retourner la réponse. La voici en action :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

// Nous n'avons plus besoin du use pour l'objet Response
// use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // On utilise le raccourci : il crée un objet Response
        // Et lui donne comme contenu le contenu du template
        return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
            'id' => $id,
        ));
    }
}

```

Et voilà, en une seule ligne, c'est bouclé ! C'est comme ça que nous générerons la plupart de nos réponses. Finalement, l'objet Response est utilisé en coulisses, nous n'avons pas à le manipuler directement.

N'oubliez pas de créer la vue associée bien entendu :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/view/Blog/voir.html.twig #-}

<!DOCTYPE html>
<html>
    <head>
        <title>Lecture de l'article {{ id }}</title>
    </head>
    <body>
        <h1>Hello Article n°{{ id }} !</h1>
    </body>
</html>

```

Si vous ne deviez retenir qu'une seule chose de cette section, c'est bien cette méthode `<?php $this->render()`, car c'est vraiment ce que nous utiliserons en permanence. 😊

Réponse et redirection

Vous serez sûrement amené à faire une redirection vers une autre page. Or notre contrôleur est *obligé* de retourner une réponse. Comment gérer une redirection ? Eh bien, vous avez peut-être évité le piège, mais *une redirection est une réponse HTTP*. Pour faire cela, il existe également un raccourci du contrôleur : la méthode `<?php $this->redirect()`. Elle prend en paramètre l'URL vers laquelle vous souhaitez faire la redirection et s'occupe de créer une réponse puis d'y définir un header qui contiendra votre URL. En action, cela donne le code suivant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // On utilise la méthode « generateUrl() » pour obtenir l'URL
        // de la liste des articles à la page 2
        // Par exemple
        return $this->redirect( $this->generateUrl('sdzblog_accueil',
array('page' => 2)) );
    }
}
```

Essayez d'aller à l'adresse http://localhost/Symfony/web/app_dev.php/blog/article/5 et vous serez redirigé vers l'accueil !

Changer le Content-type de la réponse

Lorsque vous retournez autre chose que du HTML, il faut que vous changez le Content-type de la réponse. Ce Content-type permet au navigateur qui recevra votre réponse de savoir à quoi s'attendre dans le contenu. Prenons l'exemple suivant : vous recevez une requête Ajax et souhaitez retourner un tableau en JSON :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // Créons nous-mêmes la réponse en JSON, grâce à la fonction
        json_encode()
        $response = new Response(json_encode(array('id' => $id)));

        // Ici, nous définissons le Content-type pour dire que l'on
        // renvoie du JSON et non du HTML
        $response->headers->set('Content-Type', 'application/json');

        return $response;
    }

    // Nous n'avons pas utilisé notre template ici, car il n'y en a
    // pas vraiment besoin
}
```

Testez ce rendu en allant sur http://localhost/Symfony/web/app_dev.php/blog/article/5.

Les différents services

Qu'est-ce qu'un service ?

Je vous en ai déjà brièvement parlé : un service est un script qui remplit un rôle précis et que l'on peut utiliser depuis notre contrôleur.

Imaginez par exemple un service qui a pour but d'envoyer des e-mails. Depuis notre contrôleur, on appelle ce service, on lui donne les informations nécessaires (contenu de l'e-mail, destinataire, etc.), puis on lui dit d'envoyer l'e-mail. Ainsi, toute la logique « création et envoi d'e-mail » se trouve dans ce service et non dans notre contrôleur. Cela nous permet de réutiliser ce service très facilement ! En effet, si vous codez en dur l'envoi d'e-mail dans un contrôleur A et que, plus tard, vous avez envie d'envoyer un autre e-mail depuis un contrôleur B, comment réutiliser ce que vous aviez déjà fait ? C'est impossible et c'est exactement pour cela que les services existent.



Il existe un chapitre sur les services plus loin dans ce tutoriel. N'allez pas le lire maintenant car il demande des notions que vous n'avez pas encore. Patience, sachez juste que c'est un point incontournable de Symfony2 et que nous les traiterons bien plus en détail par la suite. 😊

Accéder aux services

Pour accéder aux services depuis votre contrôleur, il faut utiliser la méthode `<?php $this->get()` du contrôleur. Par exemple, le service pour envoyer des e-mails se nomme justement « mailer ». Pour employer ce service, nous faisons donc :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // Récupération du service
        $mailer = $this->get('mailer');

        // Création de l'e-mail : le service mailer utilise
        // SwiftMailer, donc nous créons une instance de Swift_Message
        $message = \Swift_Message::newInstance()
            ->setSubject('Hello zéro !')
            ->setFrom('tutorial@symfony2.com')
            ->setTo('votre@email.com')
            ->setBody('Coucou, voici un email que vous venez de recevoir
        !');

        // Retour au service mailer, nous utilisons sa méthode « send() »
        // pour envoyer notre $message
        $mailer->send($message);

        // N'oublions pas de retourner une réponse, par exemple, une
        // page qui afficherait « L'email a bien été envoyé »
        return new Response('Email bien envoyé');
    }
}
```

Pour que l'envoi d'email fonctionne, n'oubliez pas de configurer vos paramètres si ce n'est pas déjà fait. Ouvrez le fichier `app/config/parameters.yml` pour modifier les paramètres `mailer_*`. Si vous voulez utiliser votre compte gmail :

Code : YAML

```
# app/config/parameters.yml

mailer_transport:  gmail
mailer_host:        smtp.gmail.com
mailer_user:        vous@gmail.com
mailer_password:   mdp
```

Et si vous voulez utiliser un serveur SMTP classique :

Code : YAML

```
# app/config/parameters.yml

mailer_transport:  smtp
mailer_host:        smtp.votre-serveur.fr
mailer_user:        identifiant
mailer_password:   mdp
```

Chargez la page `/blog/article/5`, et allez lire votre email !



Retenez donc la méthode `<?php $this->get('nom_du_service')` !

Brève liste des services

Maintenant que vous savez récupérer un service, encore faut-il connaître leur nom ! Et savoir les utiliser ! Ci-après est dressée une courte liste de quelques services utiles.

Templating

Templating est un service qui vous permet de gérer vos templates (vos vues, vous l'aurez compris). En fait, vous avez déjà utilisé ce service... via le raccourci `<?php $this->render!` Voici la version longue d'un `<?php $this->render('MonTemplate')` :

Code : PHP

```
<?php
// ...
public function voirAction($id)
{
    // Récupération du service
    $templating = $this->get('templating');

    // On récupère le contenu de notre template
    $contenu = $templating-
>render('SdzBlogBundle:Blog:voir.html.twig');

    // On crée une réponse avec ce contenu et on la retourne
    return new Response($contenu);
}
```

Le service Templating est utile, par exemple, pour notre e-mail de tout à l'heure. Nous avons écrit le contenu de l'e-mail en dur, ce qui n'est pas bien, évidemment. Nous devrions avoir un template pour cela. Et pour en récupérer le contenu, nous utilisons `<?php $templating->render()`. 😊

Une autre fonction de ce service qui peut servir, c'est `<?php $templating->exists('SdzBlogBundle:Blog:inexistant')` qui permet de vérifier si « SdzBlogBundle:Blog:inexistant » existe ou non.

Request

Eh oui, encore elle. C'est également un service ! C'est pour cela qu'on l'a récupéré de cette façon : `<?php $this->get('request')`.

Session

Les outils de session sont également intégrés dans un service. Vous pouvez le récupérer via `<?php $this->get('session')`. Pour définir et récupérer des variables en session, il faut utiliser les méthodes « get » et « set », tout simplement :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function voirAction($id)
    {
        // Récupération du service
        $session = $this->get('session');

        // On récupère le contenu de la variable user_id
        $user_id = $session->get('user_id');

        // On définit une nouvelle valeur pour cette variable user_id
        $session->set('user_id', 91);

        // On n'oublie pas de renvoyer une réponse
        return new Response('Désolé je suis une page de test, je n\'ai rien à dire');
    }
}
```



Pour connaître les variables de la session courante, allez dans le *profiler* (la barre d'outils en bas), rubrique **Request**, puis descendez tout en bas au paragraphe « Session Attributes ». Très utile pour savoir si vous avez bien les variables de session que vous attendez. 😊

La session se lance automatiquement dès que vous vous en servez. Voyez par exemple à la figure suivante ce que le Profiler me dit sur une page où je n'utilise pas la session.

Session Metadata

Key	Value
Created	'Thu, 01 Jan 70 01:00:00 +0100'
Last used	'Thu, 01 Jan 70 01:00:00 +0100'
Lifetime	0

Session Attributes

No session attributes

constate qu'il n'y a pas d'attribut dans la session

Et le Profiler après que nous ayons défini la variable `user_id` en session, à la figure suivante.

Session Metadata

Key	Value
Created	'Wed, 11 Jul 12 18:54:35 +0200'
Last used	'Thu, 12 Jul 12 21:46:34 +0200'
Lifetime	'0'

Session Attributes

Key	Value
<code>user_id</code>	91

Ici, on constate que l'attribut `user_id` est bien défini, avec comme valeur 91

Le Profiler nous donne même les informations sur la date de création de la session, etc.

Un autre outil très pratique du service de session est ce que l'on appelle les « messages flash ». Un terme précis pour désigner en réalité une variable de session qui ne dure que pendant 1 page. C'est une astuce utilisée pour les formulaires par exemple : la page qui traite le formulaire définit un message flash (« Article bien enregistré » par exemple) puis redirige vers la page de visualisation de l'article nouvellement créé. Sur cette page, le message flash s'affiche, et est détruit de la session. Alors si l'on change de page ou qu'on l'actualise, le message flash ne sera plus présent. Voici un exemple d'utilisation :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function voirAction($id)
    {
        return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
            'id'    => $id
        ));
    }
}
```

```
// Rajoutez cette méthode ajouterAction :
public function ajouterAction()
{
    // Bien sûr, cette méthode devra réellement ajouter l'article
    // Mais faisons comme si c'était le cas
    $this->get('session')->getFlashBag()->add('info', 'Article bien
enregistré');

    // Le « flashBag » est ce qui contient les messages flash dans
    // la session
    // Il peut bien sûr contenir plusieurs messages :
    $this->get('session')->getFlashBag()->add('info', 'Oui oui, il est
bien enregistré !');

    // Puis on redirige vers la page de visualisation de cet
    // article
    return $this->redirect( $this->generateUrl('sdzblog_voir',
array('id' => 5)) );
}
}
```

Vous pouvez voir que la méthode `ajouterAction` définit deux messages flash (appelés ici « info »). La lecture de ces messages se fait dans la vue de l'action `voirAction`, que j'ai modifié comme ceci :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

<!DOCTYPE html>
<html>
    <head>
        <title>Bienvenue sur ma première page avec le Site du Zéro
    </title>
    </head>
    <body>
        <h1>Lecture d'un article</h1>

        <p>
            {# On affiche tous les messages flash dont le nom est « info
            » #}

            {% for message in app.session.flashbag.get('info') %}
            <p>{{ message }}</p>
            {% endfor %}
            </p>

            <p>
                Ici nous pourrons lire l'article ayant comme id : {{ id }}<br
            />
                Mais pour l'instant, nous ne savons pas encore le faire, cela
                viendra !
            </p>
        </body>
    </html>
```



La variable Twig `{{ app }}` est une variable globale, disponible partout dans vos vues. Elle contient quelques variables utiles nous le verrons, dont le service `session` que nous venons d'utiliser via `{{ app.session }}`.

Essayez d'aller sur http://localhost/Symfony/web/app_dev.php/blog/ajouter, vous allez être redirigé et voir le message flash. Faites F5 et hop, il a disparu !

Sachez également que le service « `session` » est aussi accessible depuis le service « `request` ». Ainsi, depuis un contrôleur vous pouvez faire :

Code : PHP

```
<?php
$session = $this->get('request')->getSession();
```

Les autres... et les nôtres !

Il existe évidemment bien d'autres services : nous les rencontrerons au fur et à mesure dans ce cours.

Mais il existera surtout nos propres services ! En effet, la plupart des outils que nous allons créer (un formulaire, un gestionnaire d'utilisateurs personnalisé, etc.) devront être utilisés plusieurs fois. Quoi de mieux, dans ce cas, que de les définir en tant que

service ? Nous verrons cela dans la partie IV, mais sachez qu'après une petite étape de mise en place (configuration, quelques conventions), les services sont vraiment très pratiques !

Application : le contrôleur de notre blog

Construction du contrôleur

Notre blog est un bundle plutôt simple. On va mettre toutes nos actions dans un seul contrôleur « Blog ». Plus tard, nous pourrons éventuellement créer un contrôleur Tag pour manipuler les tags.

Malheureusement, on ne connaît pas encore tous les services indispensables. À ce point du cours, on ne sait pas réaliser un formulaire, manipuler les articles dans la base de données, ni même créer de vrais templates.

Pour l'heure, notre contrôleur sera donc très simple. On va créer la base de toutes les actions que l'on a mises dans nos routes. Je vous remets sous les yeux nos routes, et on enchaîne sur le contrôleur :

Code : YAML

```
# src/Sdz/BlogBundle/Resources/config/routing.yml

sdzblog_accueil:
    pattern:    /{page}
    defaults:  { _controller: SdzBlogBundle:Blog:index, page: 1 }
    requirements:
        page: \d*

sdzblog_voir:
    pattern:    /article/{id}
    defaults:  { _controller: SdzBlogBundle:Blog:voir }
    requirements:
        id: \d+

sdzblog_ajouter:
    pattern:    /ajouter
    defaults:  { _controller: SdzBlogBundle:Blog:ajouter }

sdzblog_modifier:
    pattern:    /modifier/{id}
    defaults:  { _controller: SdzBlogBundle:Blog:modifier }
    requirements:
        id: \d+

sdzblog_supprimer:
    pattern:    /supprimer/{id}
    defaults:  { _controller: SdzBlogBundle:Blog:supprimer }
    requirements:
        id: \d+
```

Et le contrôleur « Blog » :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function indexAction($page)
    {
        // On ne sait pas combien de pages il y a
        // Mais on sait qu'une page doit être supérieure ou égale à 1
        if( $page < 1 )
        {
            // On déclenche une exception NotFoundHttpException
            // Cela va afficher la page d'erreur 404 (on pourra
            personnaliser cette page plus tard, d'ailleurs)
            throw $this->createNotFoundException('Page inexisteante (page =
            '.$page.')');
        }

        // Ici, on récupérera la liste des articles, puis on la passera
        au template

        // Mais pour l'instant, on ne fait qu'appeler le template
        return $this->render('SdzBlogBundle:Blog:index.html.twig');
    }

    public function voirAction($id)
    {
```

```

// Ici, on récupérera l'article correspondant à l'id $id

return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
    'id' => $id
));

public function ajouterAction()
{
    // La gestion d'un formulaire est particulière, mais l'idée est
    // la suivante :

    if( $this->get('request')->getMethod() == 'POST' )
    {
        // Ici, on s'occupera de la création et de la gestion du
        // formulaire

        $this->get('session')->setFlashBag()->add('notice', 'Article
        bien enregistré');

        // Puis on redirige vers la page de visualisation de cet
        // article
        return $this->redirect( $this->generateUrl('sdzblog_voir',
        array('id' => 5)) );
    }

    // Si on n'est pas en POST, alors on affiche le formulaire
    return $this->render('SdzBlogBundle:Blog:ajouter.html.twig');
}

public function modifierAction($id)
{
    // Ici, on récupérera l'article correspondant à l'$id
    // Ici, on s'occupera de la création et de la gestion du
    // formulaire

    return $this->render('SdzBlogBundle:Blog:modifier.html.twig');
}

public function supprimerAction($id)
{
    // Ici, on récupérera l'article correspondant à l'$id
    // Ici, on générera la suppression de l'article en question

    return $this->render('SdzBlogBundle:Blog:supprimer.html.twig');
}
}

```

À retenir

L'erreur 404

Je vous ai donné un exemple qui vous montre comment déclencher une erreur 404. C'est quelque chose que l'on fera souvent, par exemple dès qu'un article n'existera pas, qu'un argument ne sera pas bon (page = 0), etc. Lorsque l'on déclenche cette exception, le noyau l'attrape et génère une belle page d'erreur 404. Vous pouvez aller voir le [cookbook Comment personnaliser ses pages d'erreur](#).

La définition des méthodes

Nos méthodes vont être appelées par le noyau : elles doivent donc respecter le nom et les arguments que nous avons défini dans nos routes et se trouver dans le *scope* « public ». Vous pouvez bien entendu rajouter d'autres méthodes, par exemple pour exécuter une fonction que vous réutiliserez dans deux actions différentes. Dans ce cas, vous ne devez pas les suffixer de « Action » (afin de ne pas confondre).

Testons-le

Naturellement, seules les actions `index` et `voir` vont fonctionner car nous n'avons pas créé les templates associés (ce sera fait dans le prochain chapitre). Cependant, nous pouvons voir le type d'erreur que Symfony2 nous génère. Allez sur la page de suppression d'un article, à l'adresse http://localhost/Symfony/web/app_dev.php/blog/supprimer/5. Vous pouvez voir que l'erreur est très explicite et nous permet de voir directement ce qui ne va pas. On a même les `logs` en dessous de l'erreur : on peut voir tout ce qui a fonctionné avant que l'erreur ne se déclenche. Notez par exemple le `log` n°4 :

Code : Console

```
Matched route "sdzblog_supprimer" (parameters: "_controller": "Sdz\BlogBundle\Contr
```

On voit que c'est bien la bonne route qui est utilisée, super ! On voit aussi que le paramètre « id » est bien défini à 5 par défaut : re-super !

On peut également tester notre erreur 404 générée manuellement lorsque ce paramètre « page » est à zéro. Allez sur http://localhost/Symfony/web/app_dev.php/blog/0, et admirez notre erreur. Regardez entre autres la *toolbar* (voir figure suivante).



Très pratique pour vérifier que tout est comme on l'attend !

Pour conclure

Créer un contrôleur à ce stade du cours n'est pas évident car vous ne connaissez et ne maîtrisez pas encore tous les services nécessaires. Seulement, vous avez pu comprendre son rôle et voir un exemple concret.

Rassurez-vous, dans la partie 4 du tutoriel on apprendra tout le nécessaire pour construire l'intérieur de nos contrôleurs. 😊
En attendant, rendez-vous au prochain chapitre pour en apprendre plus sur les *templates*.

Pour plus d'informations concernant les contrôleurs, n'hésitez pas à lire la [documentation officielle](#).

En résumé

- Le rôle du contrôleur est de retourner un objet Response : ceci est obligatoire !
- Le contrôleur construit la réponse en fonction des données qu'il a en entrée : paramètre de route et objet Request ;
- Le contrôleur se sert de tout ce dont il a besoin pour construire la réponse : la base de données, les vues, les différents services, etc.

Le moteur de template Twig

Les templates, ou vues, sont très intéressants. Nous l'avons déjà vu, leur objectif est de séparer le code PHP du code HTML. Ainsi, lorsque vous faites du PHP, vous n'avez pas 100 balises HTML qui gênent la lecture de votre code PHP. De même, lorsque votre designer fait du HTML, il n'a pas à subir votre code barbare PHP auquel il ne comprend rien.

Intéressé ? Lisez la suite. 😊

Les templates Twig

Intérêt

Les *templates* vont nous permettre de séparer le code PHP du code HTML/XML/Text, etc. Seulement, pour faire du HTML de présentation, on a toujours besoin d'un peu de code dynamique : faire une boucle pour afficher tous les articles d'un blog, créer des conditions pour afficher un menu différent pour les utilisateurs authentifiés ou non, etc. Pour faciliter ce code dynamique dans les templates, le moteur de template Twig offre son pseudo-langage à lui. Ça n'est pas du PHP, mais c'est plus adapté et voici pourquoi :

- la syntaxe est plus concise et plus claire. Rappelez-vous, pour afficher une variable, `{ { mavar } }` suffit, alors qu'en PHP il faudrait faire `<?php echo $mavar; ?>` ;
- il y a quelques fonctionnalités en plus, comme l'héritage de templates (nous le verrons). Cela serait bien entendu possible en PHP, mais il faudrait coder soi-même le système et cela ne serait pas aussi esthétique ;
- il sécurise vos variables automatiquement : plus besoin de se soucier de `htmlentities()`, `addslashes()` ou que sais-je encore.

 Pour ceux qui se posent la question de la rapidité : aucune inquiétude ! Oui il faut transformer le langage Twig en PHP avant de l'exécuter pour, finalement, afficher notre contenu. Mais Twig ne le fait que la première fois et met en cache du code PHP simple afin que, dès la 2^e exécution de votre page, ce soit en fait aussi rapide que du PHP simple.

Des pages Web mais aussi des e-mails et autres

En effet, pourquoi se limiter à nos pages HTML ? Les templates peuvent (et doivent) être utilisés partout. Quand on enverra des e-mails, leurs contenus seront placés dans un template. Il existe bien sûr un moyen de récupérer le contenu d'un template sans l'afficher immédiatement. Ainsi, en récupérant le contenu du template dans une variable quelconque, on pourra le passer à la fonction mail de notre choix.

Mais il en va de même pour un flux RSS par exemple ! Si l'on sait afficher une liste des news de notre site en HTML grâce au template `liste_news.html.twig`, alors on saura afficher un fichier RSS en gardant le même contrôleur, mais en utilisant le template `liste_news.rss.twig` à la place.

En pratique

On a déjà créé un template, mais un rappel ne fait pas de mal. Depuis le contrôleur, voici la syntaxe pour retourner une réponse HTTP toute faite, dont le contenu est celui d'un certain template :

Code : PHP

```
<?php
// Depuis un contrôleur

return $this->render('SdzBlogBundle:Blog:index.html.twig', array(
    'var1' => $var1,
    'var2' => $var2
));
```

Et voici comment, au milieu d'un contrôleur, récupérer le contenu d'un template en texte :

Code : PHP

```
<?php
// Depuis un contrôleur

$contenu = $this->renderView('SdzBlogBundle:Blog:email.txt.twig',
array(
    'var1' => $var1,
    'var2' => $var2
));

// Puis on envoie l'e-mail, par exemple :
mail('moi@siteduzero.com', 'Inscription OK', $contenu);
```

Et le template `SdzBlogBundle:Blog:email.txt.twig` contiendrait par exemple :

Code : Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/email.txt.twig #}

Bonjour {{ pseudo }},

Toute l'équipe du site se joint à moi pour vous souhaiter la
bienvenue sur notre site !

Revenez nous voir souvent !
```

À savoir

Première chose à savoir sur Twig : *vous pouvez afficher des variables et pouvez exécuter des expressions.* Ça n'est pas la même chose :

- {{ ... }} affiche quelque chose ;
- {% ... %} fait quelque chose ;
- {# ... #} n'affiche rien et ne fait rien : c'est la syntaxe pour les commentaires.

L'objectif de la suite de ce chapitre est donc :

- D'abord, vous donner les outils pour afficher des variables : variables simples, tableaux, objets, appliquer des filtres, etc. ;
- Ensuite, vous donner les outils pour construire un vrai code dynamique : faire des boucles, des conditions, etc. ;
- Enfin, vous donner les outils pour organiser vos templates grâce à l'héritage et à l'inclusion de templates. Ainsi vous aurez un template maître qui contiendra votre design (avec les balises <html>, <head>, etc.) et vos autres templates ne contiendront que le contenu de la page (liste des news, etc.).

Afficher des variables

Syntaxe de base pour afficher des variables

Afficher une variable se fait avec les doubles accolades « {{ ... }} ». Voici quelques exemples.

Description	Exemple Twig	Equivalent PHP
Afficher une variable	Pseudo : {{ pseudo }}	Pseudo : <?php echo \$pseudo; ?>
Afficher l'index d'un tableau	Identifiant : {{ user['id'] }}	Identifiant : <?php echo \$user['id']; ?>
Afficher l'attribut d'un objet, dont le getter respecte la convention \$objet->getAttribut()	Identifiant : {{ user.id }}	Identifiant : <?php echo \$user->getId(); ?>
Afficher une variable en lui appliquant un filtre. Ici, « upper » met tout en majuscules :	Pseudo en majuscules : {{ pseudo upper }}	Pseudo en lettre majuscules : <?php echo strtoupper(\$pseudo); ?>
Afficher une variable en combinant les filtres. « striptags » supprime les balises HTML. « title » met la première lettre de chaque mot en majuscule. Notez l'ordre d'application des filtres, ici striptags est appliqué, puis title.	Message : {{ news.texte striptags title }}	Message : <?php echo ucwords(strip_tags(\$news->getTexte())); ?>
Utiliser un filtre avec des arguments. Attention il faut que date soit un objet de type Datetime ici.	Date : {{ date date('d/m/Y') }}	Date : <?php echo \$date->format('d/m/Y'); ?>
Concaténer	Identité : {{ nom ~ " " ~ prenom }}	Identité : <?php echo \$nom.' '.\$prenom; ?>

Précisions sur la syntaxe {{ objet.attribut }}

Le fonctionnement de la syntaxe {{ objet.attribut }} est un peu plus complexe qu'il n'en a l'air. Il ne fait pas seulement objet->getAttribut. En réalité, voici ce qu'il fait exactement :

- Il vérifie si **objet** est un tableau, et si **attribut** est un index valide. Si c'est le cas, il affiche `objet['attribut']` ;
- Sinon, et si **objet** est un objet, il vérifie si **attribut** est un attribut valide (public donc). Si c'est le cas, il affiche `objet->attribut` ;
- Sinon, et si **objet** est un objet, il vérifie si `attribut()` est une méthode valide (publique donc). Si c'est le cas, il affiche `objet->attribut()` ;
- Sinon, et si **objet** est un objet, il vérifie si `getAttribut()` est une méthode valide. Si c'est le cas, il affiche `objet->getAttribut()` ;
- Sinon, et si **objet** est un objet, il vérifie si `isAttribut()` est une méthode valide. Si c'est le cas, il affiche `objet->isAttribut()` ;

- Sinon, il n'affiche rien et retourne null.

Les filtres utiles

Il y a quelques filtres disponibles nativement avec Twig, en voici quelques uns :

Filtre	Description	Exemple Twig
Upper	Met toutes les lettres en majuscules.	<code>{{ var upper }}</code>
Striptags	Supprime toutes les balises XML.	<code>{{ var striptags }}</code>
Date	Formate la date selon le format donné en argument. La variable en entrée doit être une instance de Datetime.	<code>{{ date date('d/m/Y') }}</code> Date d'aujourd'hui : {{ "now" date('d/m/Y') }}
Format	Insère des variables dans un texte, équivalent à <code>printf</code> .	<code>Il y a %s pommes et %s poires" format(153, nb_poires) }}</code>
Length	Retourne le nombre d'éléments du tableau, ou le nombre de caractères d'une chaîne.	Longueur de la variable : {{ texte length }} Nombre d'éléments du tableau : {{ tableau length }}



La documentation de tous les filtres disponibles est dans la documentation officielle de Twig :
<http://twig.sensiolabs.org/doc/filters/index.html>.

Nous pourrons également créer nos propres filtres ! On le verra plus loin dans ce cours.

Twig et la sécurité

Dans tous les exemples précédents, vos variables ont déjà été protégées par Twig ! Twig applique par défaut un filtre sur toutes les variables que vous affichez, afin de les protéger de balises HTML malencontreuses. Ainsi, si le pseudo d'un de vos membres contient un « < » par exemple, lorsque vous faites {{ pseudo }} celui-ci est échappé, et le texte généré est en réalité « mon<pseudo » au lieu de « mon<pseudo », ce qui poserait problème dans votre structure HTML. Très pratique ! Et donc à savoir : inutile de protéger vos variables en amont, Twig s'occupe de tout en fin de chaîne !

Et dans le cas où vous voulez afficher volontairement une variable qui contient du HTML (JavaScript, etc.), et que vous ne voulez pas que Twig l'échappe, il vous faut utiliser le filtre `raw` comme suit : {{ ma_variable_html|raw }}. Avec ce filtre, Twig désactivera localement la protection HTML, et affichera la variable en brut, quel que soit ce qu'elle contient.

Les variables globales

Symfony2 enregistre quelques variables globales dans Twig pour nous faciliter la vie. Voici la liste des variables globales disponibles dans tous vos templates :

Variable	Description
{{ app.request }}	Le service Request qu'on a vu au chapitre précédent sur les contrôleurs.
{{ app.session }}	Le service Session qu'on a vu également au chapitre précédent.
{{ app.environment }}	L'environnement courant : « dev », « prod », et ceux que vous avez défini.
{{ app.debug }}	True si le mode debug est activé, False sinon.
{{ app.security }}	Le service Security, que nous verrons plus loin dans ce cours.
{{ app.user }}	L'utilisateur courant, que nous verrons également plus loin dans ce cours.

Bien entendu, on peut enregistrer nos propres variables globales, pour qu'elles soient accessibles depuis toutes nos vues, au lieu de les injecter à chaque fois. Pour cela, il faut éditer le fichier de configuration de l'application comme suit :

Code : YAML

```
# app/config/config.yml
#
# ...
twig:
# ...
globals:
    webmaster: moi-même
```

Ainsi, la variable {{ webmaster }} sera injectée dans toutes vos vues, et donc utilisable comme ceci :

Code : HTML & Django

```
<footer>Responsable du site : {{ webmaster }}.</footer>
```

Je profite de cet exemple pour vous faire passer un petit message. Pour ce genre de valeurs paramétrables, la bonne pratique est de les définir non pas directement dans le fichier de configuration config.yml, mais dans le fichier des paramètres, à savoir parameters.yml. Attention je parle bien de la valeur du paramètre, non de la configuration. Voyez par vous-mêmes.

Valeur du paramètre :

Code : YAML

```
# app/config/parameters.yml

parameters:
    # ...
    app_webmaster: moi-même
```

Configuration (ici, injection dans toutes les vues) qui utilise le paramètre :

Code : YAML

```
# app/config/config.yml

twig:
    globals:
        webmaster: %app_webmaster%
```

On a ainsi séparé la valeur du paramètre, stockée dans un fichier simple ; et l'utilisation de ce paramètre, perdue dans le fichier de configuration.

Structures de contrôle et expressions

Les structures de contrôle

Nous avons vu comment *afficher* quelque chose, maintenant nous allons *faire* des choses, avec la syntaxe {{ ... }}.

Condition : {{ if }}

Exemple Twig :

Code : Django

```
{% if membre.age < 12 %}
Il faut avoir 12 ans pour ce film.
{% elseif membre.age < 18 %}
OK bon film.
{% else %}
Un peu vieux pour voir ce film non ?
{% endif %}
```

Équivalent PHP :

Code : PHP

```
<?php if($membre->getAge() < 12) { ?>
    Il faut avoir 12 ans pour ce film.
<?php } elseif($membre->getAge() < 18) { ?>
    OK bon film.
<?php } else { ?>
    Un peu vieux pour voir ce film non ?
<?php } ?>
```

Boucle : {{ for }}

Exemple Twig :

Code : HTML & Django

```
<ul>
  {%
    for membre in liste_membres %}
    <li>{{ membre.pseudo }}</li>
  {% else %}
    <li>Pas d'utilisateur trouvé.</li>
  {% endfor %}
</ul>
```

Et pour avoir accès aux clés du tableau :

Code : HTML & Django

```
<select>
  {%
    for valeur, option in liste_options %}
    <option value="{{ valeur }}">{{ option }}</option>
  {% endfor %}
</select>
```

Équivalent PHP :

Code : PHP

```
<ul>
<?php if(count($liste_membres) > 0) {
  foreach($liste_membres as $membre) {
    echo '<li>' . $membre->getPseudo() . '</li>';
  }
} else { ?>
<li>Pas d'utilisateur trouvé.</li>
<?php } ?>
</ul>
```

Avec les clés :

Code : PHP

```
<?php
foreach($liste_options as $valeur => $option) {
  // ...
}
```

Définition : {{% set %}}

Exemple Twig :

Code : Django

```
{% set foo = 'bar' %}
```

Équivalent PHP :

Code : PHP

```
<?php $foo = 'bar'; ?>
```

Une petite information sur la structure {{ for }}, celle-ci définit une variable {{ loop }} au sein de la boucle, qui contient les attributs suivants :

Variable	Description
{{ loop.index }}	Le numéro de l'itération courante (en commençant par 1).
{{ loop.index0 }}	Le numéro de l'itération courante (en commençant par 0).
{{ loop.revindex }}	Le nombre d'itérations restantes avant la fin de la boucle (en finissant par 1).
{{ loop.revindex0 }}	Le nombre d'itérations restantes avant la fin de la boucle (en finissant par 0).
{{ loop.first }}	True si c'est la première itération, false sinon.
{{ loop.last }}	True si c'est la dernière itération, false sinon.

<code>{ { loop.length } }</code>	Le nombre total d'itération dans la boucle.
----------------------------------	---

Les tests utiles

Defined

Pour vérifier si une variable existe.

Exemple Twig :

Code : Django

```
{% if var is defined %} ... {% endif %}
```

Équivalent PHP :

Code : PHP

```
<?php if(isset($var)) { }
```

Even / Odd

Pour tester si un nombre est pair / impair.

Exemple Twig :

Code : HTML & Django

```
{% for valeur in liste %}
  <span class="{% if loop.index is even %}pair{% else %}
  impair{% endif %}">
    {{ valeur }}
  </span>
{% endfor %}
```

Équivalent PHP :

Code : PHP

```
<?php
$i = 0;
foreach($liste as $valeur) {
  echo '<span class=""';
  echo $i % 2 ? 'impair' : 'pair';
  echo '">' . $valeur . '</span>';
  $i++;
}
```



La documentation de tous les tests disponibles est dans la documentation officielle de Twig.

Hériter et inclure des templates

L'héritage de template

Je vous ai fait un *teaser* précédemment : l'héritage de templates va nous permettre de résoudre la problématique : « *J'ai un seul design et n'ai pas l'envie de le répéter sur chacun de mes templates* ». C'est un peu comme ce que vous devez faire aujourd'hui avec les `include()`, mais en mieux !

Le principe

Le principe est simple : vous avez un template père qui contient le design de votre site ainsi que quelques trous (appelés *blocks* en anglais, que nous nommerons « blocs » en français) et des templates fils qui vont remplir ces blocs. Les fils vont donc venir hériter du père en remplaçant certains éléments par leur propre contenu.

L'avantage est que les templates fils peuvent modifier plusieurs blocs du template père. Avec la technique des `include()`, un template inclus ne pourra pas modifier le template père dans un autre endroit que là où il est inclus !

Les blocs classiques sont le centre de la page et le titre. Mais en fait, c'est à vous de les définir ; vous en ajouterez donc autant que vous voudrez.

La pratique

Voici à quoi peut ressembler un template père (appelé plus communément *layout*). Mettons-le dans `src/Sdz/BlogBundle/Resources/views/layout.html.twig`:

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/layout.html.twig #-}

<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block title %}SdzBlog{% endblock %}</title>
  </head>
  <body>

    {% block body %}
    {% endblock %}

  </body>
</html>
```

Voici un de nos templates fils. Mettons-le dans `src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig`:

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #-}

{% extends "SdzBlogBundle::layout.html.twig" %}

{% block title %}{{ parent() }} - Index{% endblock %}

{% block body %}
  OK, même s'il est pour l'instant un peu vide, mon blog sera trop
  bien !
{% endblock %}
```

Qu'est-ce que l'on vient de faire ?

Pour bien comprendre tous les concepts utilisés dans cet exemple très simple, détaillons un peu.

Le nom du template père

On a placé ce template dans `views/layout.html.twig` et non dans `views/qqch/layout.html.twig`. C'est tout à fait possible ! En fait, il est inutile de mettre les templates qui concernent pas un contrôleur particulier et qui peuvent être réutilisés par plusieurs contrôleurs dans un sous-répertoire. Attention à la notation pour accéder à ce template : du coup, ça n'est plus `SdzBlogBundle:MonController:layout.html.twig`, mais `SdzBlogBundle::layout.html.twig`. C'est assez intuitif, en fait : on enlève juste la partie qui correspond au répertoire `MonController`. C'est ce que l'on a fait à la première ligne du template fils.

La balise `{% block %}` côté père

Pour définir un « trou » (dit *bloc*) dans le template père, nous avons utilisé la balise `{% block %}`. Un bloc doit avoir un nom afin que le template fils puisse modifier tel ou tel bloc de façon nominative. La base, c'est juste de faire `{% block nom_du_bloc %}{% endblock %}` et c'est ce que nous avons fait pour le `body`. Mais vous pouvez insérer un texte par défaut dans les blocs, comme on l'a fait pour le `titre`. C'est utile pour deux cas de figure :

- lorsque le template fils ne redéfinit pas ce bloc. Plutôt que de n'avoir rien d'écrit, vous aurez cette valeur par défaut ;
- lorsque les templates fils veulent réutiliser une valeur commune. Par exemple, si vous souhaitez que le titre de toutes les pages de votre site commence par « SdzBlog », alors depuis les templates fils, vous pouvez utiliser `{{ parent() }}` qui permet d'utiliser le contenu par défaut du bloc côté père. Regardez, nous l'avons fait pour le titre dans le template fils.

La balise `{% block %}` côté fils

Elle se définit exactement comme dans le template père sauf que cette fois-ci, on y met notre contenu. Mais étant donné que les blocs se définissent et se remplissent de la même façon, vous avez pu deviner qu'on peut hériter en cascade ! En effet, si l'on crée un troisième template petit-fils qui hérite de fils, on pourra faire beaucoup de choses.

Le modèle « triple héritage »

Pour bien organiser ses templates, une bonne pratique est sortie du lot. Il s'agit de faire de l'héritage de template sur trois niveaux, chacun des niveaux remplissant un rôle particulier. Les trois templates sont les suivants :

- layout général : c'est le design de votre site, indépendamment de vos bundles. Il contient le header, le footer, etc. La structure de votre site donc (c'est notre template père) ;
- layout du bundle : il hérite du layout général et contient les parties communes à toutes les pages d'un même bundle. Par exemple, pour notre blog, on pourrait afficher un menu particulier, rajouter « Blog » dans le titre, etc. ;
- template de page : il hérite du layout du bundle et contient le contenu central de votre page.

Nous verrons un exemple de ce triple héritage juste après dans l'exemple du blog.

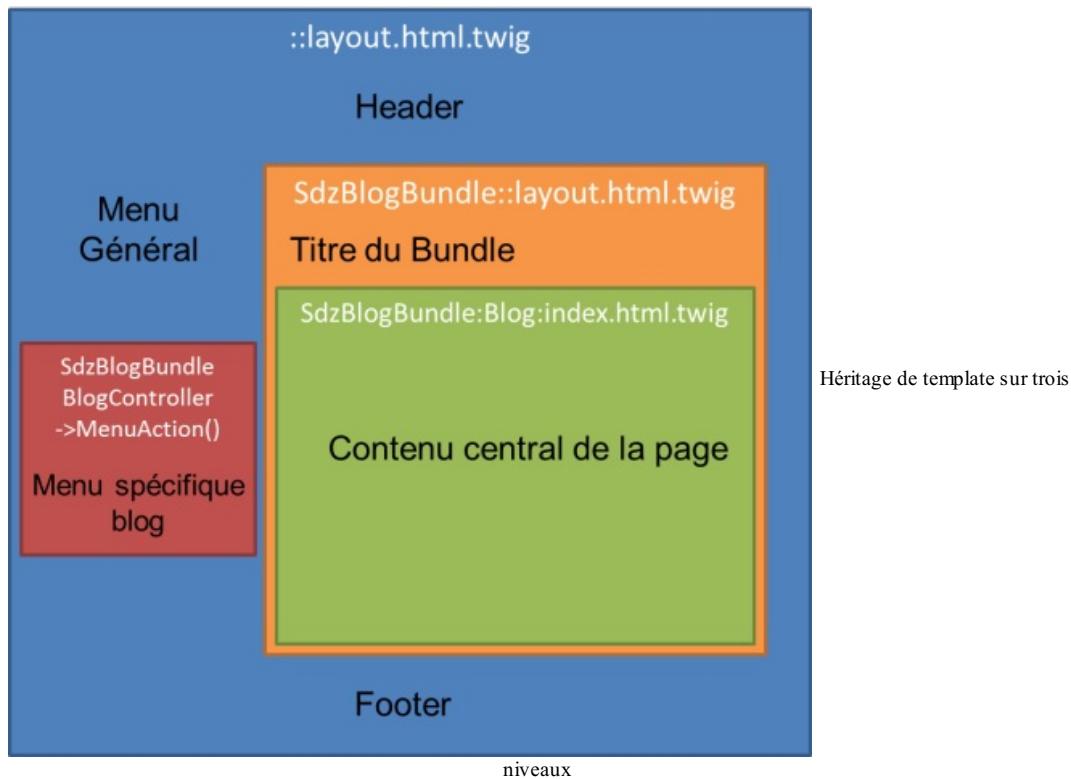


Question : puisque le layout général ne dépend pas d'un bundle en particulier, où le mettre ?

Dans votre répertoire app/ ! En effet, dans ce répertoire, vous pouvez toujours avoir des fichiers qui écrasent ceux des bundles ou bien des fichiers communs aux bundles. Le layout général de votre site fait partie de ces ressources communes. Son répertoire exact doit être app/Resources/views/layout.html.twig.

Et pour l'appeler depuis vos templates, la syntaxe est la suivante : « ::layout.html.twig ». Encore une fois, c'est très intuitif : après avoir enlevé le nom du contrôleur tout à l'heure, on enlève juste cette fois-ci le nom du bundle.

Afin de bien vous représenter l'architecture adoptée, je vous propose un petit schéma à la figure suivante. Il vaut ce qu'il vaut, mais vous permet de bien comprendre ce qu'on fait.



Je vous parle du bloc rouge un peu après, c'est une inclusion non pas de template, mais d'action de contrôleur ! Il ne fait pas parti du modèle triple héritage à proprement parler.

L'inclusion de templates

La théorie : quand faire de l'inclusion ?

Hériter, c'est bien, mais inclure, cela n'est pas mal non plus. Prenons un exemple pour bien faire la différence.

Le formulaire pour ajouter un article est le même que celui pour... modifier un article. On ne va pas faire du copier-coller de code, cela serait assez moche, et puis nous sommes fainéants. C'est ici que l'inclusion de templates intervient. On a nos deux templates SdzBlogBundle:Blog:ajouter.html.twig et SdzBlogBundle:Blog:modifier.html.twig qui héritent chacun de SdzBlogBundle::layout.html.twig. L'affichage exact de ces deux templates diffère un peu, mais chacun d'eux inclut SdzBlogBundle:Blog:formulaire.html.twig à l'endroit exact pour afficher le formulaire.

On voit bien qu'on ne peut pas faire d'héritage sur le template formulaire.html.twig car il faudrait le faire hériter une fois de ajouter.html.twig, une fois de modifier.html.twig, etc. Comment savoir ? Et si un jour nous souhaitons ne le

faire hériter de rien du tout pour afficher le formulaire tout seul dans une *popup* par exemple ? Bref, c'est bien une inclusion qu'il nous faut ici.

La pratique : comment le faire ?

Comme toujours avec Twig, cela se fait très facilement. Il faut utiliser la balise `{% include %}`, comme ceci :

Code : HTML & Django

```
{% include "SdzBlogBundle:Blog:formulaire.html.twig" %}
```

Ce code inclura le contenu du template à l'endroit de la balise. Une sorte de copier-coller automatique, en fait ! Voici un exemple avec la vue `ajouter.html.twig` :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/ajouter.html.twig #}

{% extends "SdzBlogBundle::layout.html.twig" %}

{% block body %}

<h2>Ajouter un article</h2>

{% include "SdzBlogBundle:Blog:formulaire.html.twig" %}

<p>
    Attention : cet article sera ajouté directement
    sur la page d'accueil après validation du formulaire.
</p>

{% endblock %}
```

Et voici le code du template inclus :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/formulaire.html.twig #}

{# Cette vue n'hérite de personne, elle sera inclue par d'autres
vues qui, elles, hériteront sûrement du layout. #}
{# Je dis « sûrement » car, ici pour cette vue, on n'en sait rien
et c'est une info qui ne nous concerne pas. #}

<h3>Formulaire d'article</h3>

{# Ici on laisse vide la vue pour l'instant, on la comblera plus
tard lorsqu'on saura afficher un formulaire. #}
<div class="well">
    Ici se trouvera le formulaire.
</div>
```



À l'intérieur du template inclus, vous retrouvez toutes les variables qui sont disponibles dans le template qui fait l'inclusion : exactement comme si vous copiez-collez le contenu.

L'inclusion de contrôleur

La théorie : quand inclure des contrôleur ?

Voici un dernier point à savoir absolument avec Twig, un des points les plus puissants dans son utilisation avec Symfony2. On vient de voir comment inclure des templates : ceux-ci profitent des variables du template qui fait l'inclusion, très bien.

Seulement dans bien des cas, depuis le template qui fait l'inclusion, vous voudrez inclure un autre template mais vous n'avez pas les variables nécessaires pour lui. Restons sur l'exemple de notre blog, dans le schéma précédent je vous ai mis un bloc rouge : considérons que dans cette partie du menu, accessible sur toutes les pages même hors du blog, on veut afficher les 3 derniers articles du blog.

C'est donc depuis le layout général qu'on va inclure non pas un template du bundle Blog — nous n'aurions pas les variables à lui donner — mais un contrôleur du bundle Blog. Le contrôleur va créer les variables dont il a besoin, et les donner à son template, pour ensuite être inclus là où on le veut !

La pratique : comment le faire ?

Au risque de me répéter : cela se fait très simplement !

Du côté du template qui fait l'inclusion, à la place de la balise `{% include %}`, il faut utiliser le code suivant :

Code : Autre

```
{% render "SdzBlogBundle:Blog:menu" %}
```

Ici, `SdzBlogBundle:Blog:menu` n'est pas un template mais une action de contrôleur, c'est la syntaxe qu'on utilise dans les routes, vous l'aurez reconnu.

Voici par exemple ce qu'on mettrait dans le layout :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #}
<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block title %}SdzBlog{% endblock %}</title>
  </head>
  <body>

    <div id="menu">
      {% render "SdzBlogBundle:Blog:menu" %}
    </div>

    {% block body %}
    {% endblock %}

  </body>
</html>
```

Et du côté du contrôleur, c'est une méthode très classique (regardez la ligne 16) :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...

public function menuAction()
{
    // On fixe en dur une liste ici, bien entendu par la suite on
    // la récupérera depuis la BDD !
    $liste = array(
        array('id' => 2, 'titre' => 'Mon dernier weekend !'),
        array('id' => 5, 'titre' => 'Sortie de Symfony2.1'),
        array('id' => 9, 'titre' => 'Petit test')
    );

    return $this->render('SdzBlogBundle:Blog:menu.html.twig', array(
        'liste_articles' => $liste // C'est ici tout l'intérêt : le
        // contrôleur passe les variables nécessaires au template !
    ));
}
```

Et enfin, un exemple de ce que pourrait être le template `menu.html.twig`:

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/menu.html.twig #}

{# Ce template n'hérite de personne, tout comme le template inclus
avec {% include %}. #}

<ul>
<ul class="nav nav-pills nav-stacked">
  {% for article in liste_articles %}
    <li><a href="{{ path('sdzblog_voir', { 'id': article.id}) }}>{{ article.titre }}</a></li>
  {% endfor %}
```

```
</ul>
</ul>
```

Application : les templates de notre blog

Revenons à notre blog. Faites en sorte d'avoir sous la main le contrôleur que l'on a réalisé au chapitre précédent. Le but ici est de créer tous les templates que l'on a utilisés depuis le contrôleur, ou du moins leur squelette. Étant donné que l'on n'a pas encore la vraie liste des articles, on va faire avec des variables vides : ça va se remplir par la suite, mais le fait d'employer des variables vides va nous permettre dès maintenant de construire le template.

Pour encadrer tout ça, nous allons utiliser le modèle d'héritage sur trois niveaux : layout général, layout du bundle et template.

Layout général

La théorie

Comme évoqué précédemment, le layout est la structure HTML de notre site avec des blocs aux endroits stratégiques pour permettre aux templates qui hériteront de ce dernier de personnaliser la page. On va ici créer une structure simple ; je vous laisse la personnaliser si besoin est. Pour les blocs, pareil pour l'instant, on fait simple : un bloc pour le body et un bloc pour le titre.

Je vais également en profiter pour introduire l'utilisation de ressources CSS/JS/etc. dans Symfony2. Cela se fait très bien avec la fonction `{ { asset() } }` de Twig, qui va chercher vos ressources dans le répertoire `/web`. Regardez simplement comment elle est utilisée dans l'exemple et vous saurez l'utiliser de façon basique.

 Pour le design du blog que l'on va construire, je vais utiliser le [bootstrap de Twitter](#). C'est un framework CSS, l'équivalent pour le CSS de ce que Symfony2 est pour le PHP. Cela permet de faire des beaux designs, boutons ou liens très rapidement. Vous pourrez le voir dans les vues que je fais par la suite. Mais tout d'abord vous devez [le télécharger](#) et l'extraire dans le répertoire `/web`. Vous devez avoir le fichier `bootstrap.css` disponible dans le répertoire `/web/css/bootstrap.css` par exemple. Il existe [un cours sur Bootstrap](#) sur le Site du Zéro. Je vous invite à y jeter un œil si ce framework vous intéresse, ou juste par curiosité.

La pratique

Commençons par faire le layout général de l'application, la vue située dans le répertoire `/app`. Voici le code exemple que je vous propose :

Code : HTML & Django

```
{# app/Resources/views/layout.html.twig #-}

<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

    <title>{ % block title %}Sdz{ % endblock %}</title>

    { % block stylesheets %}
      <link rel="stylesheet" href="{{ asset('css/bootstrap.css') }}" type="text/css" />
    { % endblock %
  </head>

  <body>
    <div class="container">
      <div id="header" class="hero-unit">
        <h1>Mon Projet Symfony2</h1>
        <p>Ce projet est propulsé par Symfony2, et construit grâce au tutoriel du siteduzero.</p>
        <p><a class="btn btn-primary btn-large" href="http://www.siteduzero.com/tutoriel-3-517569-symfony2.html">Lire le tutoriel </a></p>
      </div>

      <div class="row">
        <div id="menu" class="span3">
          <h3>Le blog</h3>
          <ul class="nav nav-pills nav-stacked">
            <li><a href="{{ path('sdzblob_accueil') }}>Accueil du blog</a></li>
            <li><a href="{{ path('sdzblob_ajouter') }}>Ajouter un article</a></li>
          </ul>
        { % render "SdzBlogBundle:Blog:menu" with { 'nombre': 3 } %
      </div>
      <div id="content" class="span9">
        { % block body %
        { % endblock %
      </div>
    </div>
```

```

<hr>

<footer>
  <p>The sky's the limit © 2012 and beyond.</p>
</footer>
</div>

{%
  block javascripts %
  {# Ajoutez ces javascripts si vous comptez vous servir des
fonctionnalités du bootstrap Twitter #}
  <script
src="//ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
  <script type="text/javascript" src="{{ asset('js/bootstrap.js') }}></script>
{%
  endblock %

</body>
</html>

```

Voici les lignes qui contiennent un peu de Twig :

- Ligne 8 : création du bloc « title » avec « Sdz » comme contenu par défaut ;
- Lignes 11 : appel du CSS situé dans /web/css/bootstrap.css ;
- Lignes 29 et 30 : utilisation de la fonction {{ path }} pour faire des liens vers d'autres routes ;
- Ligne 33 : inclusion de la méthode menu du contrôleur Blog du bundle SdzBlogBundle, avec l'argument nombre défini à 3 ;
- Lignes 36 et 37 : création du bloc « body » sans contenu par défaut.

Et voilà, nous avons notre layout général ! Pour pouvoir tester nos pages, il faut maintenant s'attaquer au layout du bundle.

Layout du bundle

La théorie

Comme on l'a dit, ce template va hériter du layout général, rajouter la petite touche perso au bundle Blog, puis se faire hériter à son tour par les templates finaux. En fait, il ne contient pas grand-chose. Laissez courir votre imagination, mais moi, je ne vais rajouter qu'une balise <h1>, vous voyez ainsi le mécanisme et pouvez personnaliser à votre sauce.

La seule chose à laquelle il faut faire attention, c'est au niveau du nom des blocs que ce template crée pour ceux qui vont l'hériter. Une bonne pratique consiste à préfixer le nom des blocs par le nom du bundle courant. Regardez le code et vous comprendrez.

La pratique

Voici ce que j'ai mis pour le layout du bundle :

Code : HTML & Django

```

{# src/Sdz/Bundle/Resources/views/layout.html.twig #-}

{%
  extends "::layout.html.twig" %

  % block title %
    Blog - {{ parent() }}
  {%
    endblock %

  % block body %
    {# On définit un sous-titre commun à toutes les pages du bundle,
    par exemple #}
    <h1>Blog</h1>

    <hr>

    {# On définit un nouveau block, que les vues du bundle pourront
    remplir #}
    {%
      block sdzblob_body %
    {%
      endblock %

  {%
    endblock %
}
```

On a ajouté un <h1> dans le bloc body puis créé un nouveau bloc qui sera personnalisé par les templates finaux. On a préfixé le nom du nouveau bloc pour le body afin d'avoir un nom unique pour notre bundle.

Les templates finaux

La théorie

Pas trop de théorie ici, il ne reste plus qu'à hériter et personnaliser les templates.

Blog/index.html.twig

C'est le template de la page d'accueil. On va faire notre première boucle sur la variable `articles`. Cette variable n'existe pas encore, on va modifier le contrôleur juste après.

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #}

{% extends "SdzBlogBundle::layout.html.twig" %}

{% block title %}
    Accueil - {{ parent() }}
{% endblock %}

{% block sdzblog_body %}

    <h2>Liste des articles</h2>

    <ul>
        {% for article in articles %}
            <li>
                <a href="{{ path('sdzblog_voir', {'id': article.id}) }}">{{ article.titre }}</a>
                par {{ article.auteur }},  

                le {{ article.date|date('d/m/Y') }}
            </li>
        {% else %}
            <li>Pas (encore !) d'articles</li>
        {% endfor %}
    </ul>
{% endblock %}
```

Pas grand-chose à dire, on a juste utilisé les variables et expressions expliquées dans ce chapitre.

Afin que cette page fonctionne, il nous faut modifier l'action `indexAction()` du contrôleur pour passer une variable `articles` à cette vue. Pour l'instant, voici juste de quoi se débarrasser de l'erreur :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// Dans l'action indexAction() :
return $this->render('SdzBlogBundle:Blog:index.html.twig', array(
    'articles' => array()
));
```



Vous pouvez dès maintenant voir votre nouvelle peau : http://localhost/Symfony/web/app_dev.php/blog !

Si vous n'aviez pas rajouté l'action `menu` du contrôleur tout à l'heure, voici comment le faire, et aussi comment l'adapter à l'argument qu'on lui a passé cette fois-ci :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function menuAction($nombre) // Ici, nouvel argument
$nombre, on a transmis via le with depuis la vue
{
    // On fixe en dur une liste ici, bien entendu par la suite on
    // la récupérera depuis la BDD !
    // On pourra récupérer $nombre articles depuis la BDD, avec
    // $nombre un paramètre qu'on peut changer lorsqu'on appelle cette
    // action
    $liste = array(
        array('id' => 2, 'titre' => 'Mon dernier weekend !'),
        array('id' => 5, 'titre' => 'Sortie de Symfony2.1'),
        array('id' => 9, 'titre' => 'Petit test')
    );
}
```

```

    return $this->render('SdzBlogBundle:Blog:menu.html.twig', array(
        'liste_articles' => $liste // C'est ici tout l'intérêt : le
        contrôleur passe les variables nécessaires au template !
    ));
}

```

Avec sa vue associée :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/menu.html.twig #-}

<h3>Les derniers articles</h3>

<ul class="nav nav-pills nav-stacked">
    {% for article in liste_articles %}
        <li><a href="{{ path('sdzblog_voir', { 'id': article.id}) }}>{{
            article.titre }}</a></li>
    {% endfor %}
</ul>

```

Si vous avez bien ajouté le CSS de Twitter, le résultat devrait ressembler à la figure suivante.

Mon Projet Symfony2

Ce projet est propulsé par Symfony2, et construit grâce au tutoriel du siteduzero.

[Lire le tutoriel »](#)

Le blog

[Accueil du blog](#)

[Ajouter un article](#)

Les derniers articles

[Mon dernier weekend !](#)

[Sortie de Symfony2.1](#)

[Petit test](#)

Blog

Liste des articles

- [Mon weekend a Phi Phi Island !](#) par winzou, le 18/07/2012
- [Répetition du National Day de Singapour](#) par winzou, le 18/07/2012
- [Chiffre d'affaire en hausse](#) par M@teo21, le 18/07/2012

The sky's the limit © 2012 and beyond.

Le rendu de notre blog

Vous voulez voir des articles au lieu du message pas très drôle ? Voici un tableau d'articles à ajouter temporairement dans la méthode `indexAction()`, que vous pouvez passer en paramètre à la méthode `render()`. C'est un tableau pour l'exemple, par la suite il faudra bien sûr récupérer les articles depuis la base de données.

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...

public function indexAction()
{
    // ...

    // Les articles :
    $articles = array(
        array('titre' => 'Mon weekend a Phi Phi Island !', 'id' => 1, 'auteur' => 'winzou', 'contenu' => 'Ce weekend était trop bien. Blabla...', 'date' => new \Datetime()),
        array('titre' => 'Répetition du National Day de Singapour', 'id' => 2, 'auteur' => 'winzou', 'contenu' => 'Bientôt prêt pour le jour J. Blabla...', 'date' => new \Datetime()),
        array('titre' => 'Chiffre d\'affaire en hausse', 'id' => 3, 'auteur' => 'M@teo21', 'contenu' => '+500% sur 1 an, fabuleux.')
    );
}

```

```

    Blabla...',      'date' => new \Datetime()
);

// Puis modifiez la ligne du render comme ceci, pour prendre en
// compte nos articles :
return $this->render('SdzBlogBundle:Blog:index.html.twig', array(
    'articles' => $articles
));
}

```

Rechargez la page, et profitez du résultat.



Attention, on vient de définir des articles en brut dans le contrôleur, mais c'est uniquement pour l'exemple d'utilisation de Twig ! Ce n'est bien sûr pas du tout une façon correcte de le faire, par la suite nous les récupérerons depuis la base de données.

Blog/voir.html.twig

Il ressemble beaucoup à `index.html.twig` sauf qu'on passe à la vue une variable `{{ article }}` contenant un seul article, et non plus une liste d'articles. Voici un code par exemple :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

{% extends "SdzBlogBundle::layout.html.twig" %}

{% block title %}
    Lecture d'un article - {{ parent() }}
{% endblock %}

{% block sdzblog_body %}

    <h2>{{ article.titre }}</h2>
    <i>Par {{ article.auteur }}, le {{ article.date|date('d/m/Y') }}</i>

    <div class="well">
        {{ article.contenu }}
    </div>

    <p>
        <a href="{{ path('sdzblog_accueil') }}" class="btn">
            <i class="icon-chevron-left"></i>
            Retour à la liste
        </a>
        <a href="{{ path('sdzblog_modifier', { 'id': article.id }) }}" class="btn">
            <i class="icon-edit"></i>
            Modifier l'article
        </a>
        <a href="{{ path('sdzblog_supprimer', { 'id': article.id }) }}" class="btn">
            <i class="icon-trash"></i>
            Supprimer l'article
        </a>
    </p>
{% endblock %}

```

Et l'adaptation du contrôleur bien évidemment :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...

public function voirAction($id)
{
    // ...

    $article = array(
        'id'      => 1,
        'titre'   => 'Mon weekend a Phi Phi Island !',
        'auteur'  => 'winzou',
        'contenu' => 'Ce weekend était trop bien. Blabla...',

```

```

    'date'      => new \Datetime()
);

// Puis modifiez la ligne du render comme ceci, pour prendre en
// compte l'article :
return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
    'article' => $article
));
}

```

La figure suivante représente le rendu de /blog/article/1.



Le blog

[Accueil du blog](#)
[Ajouter un article](#)

Les derniers articles

[Mon dernier weekend !](#)
[Sortie de Symfony2.1](#)
[Petit test](#)

Blog

Mon weekend a Phi Phi Island !

Par winzou, le 18/07/2012

Ce weekend était trop bien. Blabla...

[Retour à la liste](#) [Modifier l'article](#) [Supprimer l'article](#)

The sky's the limit © 2012 and beyond.

Visualisation d'un article

[Blog/modifier.html.twig et ajouter.html.twig](#)

Ceux-ci contiennent une inclusion de template. En effet, rappelez-vous, j'avais pris l'exemple d'un formulaire utilisé pour l'ajout mais également la modification. C'est notre cas ici, justement. Voici donc le fichier modifier.html.twig :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/modifier.html.twig #-}

{%
    extends "SdzBlogBundle::layout.html.twig"
}

{%
    block title %}
    Modifier un article - {{ parent() }}
{%
    endblock %}

{%
    block sdzblog_body %}
    <h2>Modifier un article</h2>
    {% include "SdzBlogBundle:Blog:formulaire.html.twig" %}

    <p>
        Vous éditez un article déjà existant,
        ne le changez pas trop pour éviter
        aux membres de ne pas comprendre
        ce qu'il se passe.
    </p>

    <p>
        <a href="{{ path('sdzblog_voir', {'id': article.id}) }}" class="btn">
            <i class="icon-chevron-left"></i>
            Retour à l'article
        </a>
    </p>
{%
    endblock %}

```

Le template `ajouter.html.twig` lui ressemble énormément, je vous laisse donc le faire.

Quant à `formulaire.html.twig`, on ne sait pas encore le faire car il demande des notions de formulaire, mais faisons déjà sa structure pour le moment :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/formulaire.html.twig #}

{# Cette vue n'hérite de personne, elle sera inclue par d'autres
vues qui, elles, hériteront sûrement du layout #}
{# Je dis « sûrement » car, ici pour cette vue, on n'en sait rien
et c'est une info qui ne nous concerne pas #}

<h3>Formulaire d'article</h3>

{# Ici on laisse vide la vue pour l'instant, on la comblera plus
tard lorsque saura afficher un formulaire #}
<div class="well">
    Ici se trouvera le formulaire.
</div>
```

Une chose importante ici : dans ce template, il n'y a aucune notion de bloc, d'héritage, etc. Ce template est un électron libre : vous pouvez l'inclure depuis n'importe quel autre template.

Et bien sûr il faut adapter le contrôleur pour passer la variable article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function modifierAction($id)
{
    // Ici, on récupérera l'article correspondant à l'$id

    // Ici, on s'occupera de la création et de la gestion du
    formulaire

    $article = array(
        'id'      => 1,
        'titre'   => 'Mon weekend a Phi Phi Island !',
        'auteur'  => 'winzou',
        'contenu' => 'Ce weekend était trop bien. Blabla...',
        'date'    => new \Datetime()
    );

    // Puis modifiez la ligne du render comme ceci, pour prendre en
    compte l'article :
    return $this->render('SdzBlogBundle:Blog:modifier.html.twig',
array(
    'article' => $article
));
}
```

Ainsi, `/blog/modifier/1` nous donnera la figure suivante.

Mon Projet Symfony2

Ce projet est propulsé par Symfony2, et construit grâce au tutoriel du siteduzero.

Lire le tutoriel »

Le blog

[Accueil du blog](#)

[Ajouter un article](#)

Les derniers articles

[Mon dernier weekend !](#)

[Sortie de Symfony2.1](#)

[Petit test](#)

Blog

Modifier un article

Formulaire d'article

Ici se trouvera le formulaire.

Vous éditez un article déjà existant, ne le changez pas trop pour éviter aux membres de ne pas comprendre ce qu'il se passe.

[◀ Retour à l'article](#)

The sky's the limit © 2012 and beyond.

Modification d'un article

Pour conclure

Et voilà, nous avons générés presque tous nos templates. Bien sûr, ils sont encore un peu vides car on ne sait pas utiliser les formulaires ni récupérer les articles depuis la base de données. Mais vous savez maintenant les réaliser et c'était une étape importante ! Je vais vous laisser créer les templates manquants ou d'autres afin de vous faire la main. Bon code !

Ceci termine ce chapitre : vous savez afficher avec mise en forme le contenu de votre site. Vous avez maintenant presque toutes les billes en main pour réaliser un site internet. Bon OK, c'est vrai, il vous manque encore des concepts clés tels que les formulaires, la base de données, etc. Mais vous maîtrisez pleinement la base du framework Symfony2, et apprendre ces prochains concepts sera bien plus facile !

Pour plus d'informations concernant Twig et ses possibilités, n'hésitez pas à lire la [documentation officielle](#).

En résumé

- Un moteur de templates tel que Twig permet de bien séparer le code PHP du code HTML, dans le cadre de l'architecture MVC ;
- La syntaxe `{ { var } }` affiche la variable `var` ;
- La syntaxe `{ % if %}` fait quelque chose, ici une condition ;
- Twig offre un système d'héritage (*via { % extends % }*) et d'inclusion (*via { % include %} et { % render %}*) très intéressant pour bien organiser les templates ;
- Le modèle triple héritage est très utilisé pour des projets avec Symfony2.

Installer un bundle grâce à Composer

Je fais une parenthèse pour vous présenter un chapitre sur l'outil Composer.

Cet outil ne fait absolument pas parti de Symfony2, mais son usage est tellement omniprésent dans la communauté Symfony2 que je me dois de vous le présenter. Nous faisons donc une pause dans le développement de notre blog pour parler un peu de cet outil de gestion de dépendance, qui va nous servir à installer des bundles et autres librairies très facilement.

Bonne lecture !

Composer, qu'est-ce que c'est ?

Un gestionnaire de dépendances

Composer est un outil pour gérer les dépendances en PHP. Les dépendances dans un projet, ce sont toutes les librairies dont votre projet dépend pour fonctionner. Par exemple, votre projet utilise la librairie SwiftMailer pour envoyer des emails, il « dépend » donc de SwiftMailer. Autrement dit, SwiftMailer est une dépendance dans votre projet.

Composer a donc pour objectif de vous aider à gérer toutes vos dépendances. En effet, il y a plusieurs problématiques lorsqu'on utilise des librairies externes :

- Ces librairies sont mises à jour. Il vous faut donc les mettre à jour une à une pour vous assurer de corriger les bogues, etc. de chacune d'entre elles.
- Ces librairies peuvent elles-mêmes dépendre d'autres librairies. En effet, si une de vos librairies dépend d'autres librairies, cela vous oblige à gérer l'ensemble de ces dépendances (installation, mises à jour, etc.).
- Ces librairies ont chacune leur paramètres d'*autoload*, et vous devez gérer leur *autoload* pour chacune d'entre elles.

Composer va nous aider dans chacune de ces tâches.

Comment Composer sait où trouver les librairies ?

Très bonne question. En effet il est évident que ce système de gestion ne peut marcher que si on peut centraliser les informations de chaque librairie. Il s'agit du site www.packagist.org.

Par exemple, voici la page pour la librairie Symfony2 (et oui, c'est une librairie comme une autre !) : <https://packagist.org/packages/symfony/symfony>. Vous pouvez voir les informations comme le mainteneur principal, le site de la librairie, etc. Mais ce qui nous importe le plus, ce sont les sources ainsi que les dépendances (dans `Requires`).

Composer va donc lire ces informations, et a alors toutes les cartes en main pour télécharger Symfony2 ainsi que ses dépendances.

Un outil innovant... dans l'écosystème PHP

Ce genre d'outil de gestion de dépendances n'est pas du tout nouveau dans le monde informatique. Vous connaissez peut-être déjà APT, le gestionnaire de paquets de la distribution Linux Debian. Il existe également des outils de ce genre pour le langage Ruby par exemple. Cependant, jusque très récemment, il n'existeait aucun outil de ce genre pour PHP. La forte communauté qui s'est construite autour de Symfony2 a fait naître le besoin d'un tel outil, et l'a ensuite développé.

Concrètement, comment ça marche ?

Concrètement, voici comment s'utilise Composer :

- On définit dans un fichier la liste des librairies dont le projet dépend, ainsi que leur version ;
- On exécute une commande pour installer ou mettre à jour ces librairies (et leurs propres dépendances donc) ;
- On inclut alors le fichier *d'autoload* généré par Composer dans notre projet.

Mais avant de manipuler Composer, commençons par l'installer !

Installer Composer et Git

Installer Composer

Installer Composer est très facile, il suffit d'une seule commande... PHP ! Exécutez cette commande dans la console :

Code : Console

```
C:\wamp\www> php -r "eval('?
>'.file_get_contents('http://getcomposer.org/installer'));"
```

Cette commande va télécharger et exécuter le fichier PHP suivant : <http://getcomposer.org/installer>. Vous pouvez aller le voir, ce n'est pas Composer en lui-même mais son installateur. Il fait quelques vérifications (version de PHP, etc.), puis télécharge effectivement Composer dans le fichier `composer.phar`.

Composer en lui-même c'est le fichier PHAR `composer.phar`, c'est lui que nous devrons exécuter par la suite. Vous pouvez déjà l'exécuter pour vérifier que tout est OK :

Code : Console

```
C:\wamp\www>php composer.phar --version
```

```
Composer version a5eaba8
```

N'hésitez pas à mettre à jour Composer lui-même de temps en temps. Il faut pour cela utiliser la commande "self-update" de Composer, comme suit :

Code : Console

```
C:\wamp\www>php composer.phar self-update
Updating to version ded485d.
Downloading: 100%
```

Et voilà, je viens de mettre à jour Composer très simplement !

Cependant, l'installation n'est pas finie. En effet, pour récupérer certaines librairies, Composer utilise Git.

Installer Git

Comme on l'a vu, pour récupérer les librairies Composer se base sur les informations répertoriées sur Packagist. Si pour certaines librairies Composer peut télécharger directement des archives contenant les sources, pour d'autres il doit utiliser un gestionnaire de versions tel que Git.

En réalité beaucoup de librairies sont dans ce cas, c'est pourquoi l'installation de Git ne peut être évitée. Ce n'est pas grave, attaquons-la ! Je vais ici décrire rapidement son installation, mais sachez qu'un cours du Site du Zéro détaille très bien son fonctionnement et son installation.

Installer Git sous Windows

Sous Windows, il faut utiliser [msysgit](#). Cela installe msys (un système d'émulation des commandes Unix sous Windows) et Git lui-même.

Téléchargez le fichier et exécutez-le, cela va tout installer. Laissez les paramètres par défaut, ils conviennent très bien. Cela va prendre un peu de temps car il y a pas mal à télécharger (une centaine de Mo) et à exécuter, ne vous inquiétez pas. Une fois que vous avez une ligne de commande (/dev), vous pouvez fermer la fenêtre.

Une fois installé, il faut rajouter les exécutables Git au PATH de Windows. Rajoutez donc ceci : « ;C:\msysgit\bin;C:\msysgit\mingw\bin » à la suite de votre variable d'environnement système PATH (on en a déjà parlé lors du [premier chapitre](#)).

Redémarrez votre ordinateur, et ensuite vérifiez l'installation en exécutant la commande suivante :

Code : Console

```
C:\wamp\www>git version
git version 1.7.9.msysgit.0
```

Si vous n'avez pas d'erreur, c'est tout bon !

Installer Git sous Linux

Sous linux c'est encore plus simple avec votre gestionnaire de paquets. Voici comment l'installer depuis la distribution Debian et ses dérivés (Ubuntu, etc.) :

Code : Console

```
sudo apt-get install git-core
```

Installer un bundle grâce à Composer

Manipulons Composer

Avant d'utiliser Composer dans notre projet Symfony2, on va d'abord s'amuser avec lui sur un projet test afin de bien comprendre son fonctionnement. Créez donc un répertoire `test` là où vous avez téléchargé Composer.

Déclarer ses dépendances

La première chose à faire dans un projet, c'est de déclarer ses dépendances. Cela se fait via un fichier `composer.json`, qui contient les informations sur les librairies dont dépend votre projet ainsi que leur version. La syntaxe est assez simple, en JSON, créez le fichier suivant :

Code : JavaScript

```
// test/composer.json
{
    "require": {
        "twig/extensions": "dev-master"
    }
}
```

Ce tableau JSON est le minimum syndical : il ne précise que les dépendances via la clé `require`. Il n'y a ici qu'une seule dépendance : "twig/extensions". La version requise pour cette dépendance est "dev-master", cela signifie qu'on veut la version la plus à jour possible.

Un point sur les versions, voici ce que vous pouvez mettre :

Valeur	Exemple	Description
Un numéro de version exact	"2.0.17"	Ainsi, Composer téléchargera cette version exacte.
Une plage de version	Ainsi, Composer téléchargera la version la plus à jour, à partir de la version 2.0 et en s'arrêtant avant la version 3.0. Par exemple, si les dernières versions sont 2.9, 3.0 et 3.1, Composer va télécharger la version 2.9.	
Un numéro de version avec joker « * »	"2.0.*"	Ainsi, Composer téléchargera la version la plus à jour qui commence par 2.0. Par exemple, il téléchargerait la version 2.0.17 mais pas la version 2.1.1. C'est la façon la plus utilisée pour définir la version des dépendances.

Dans notre cas, "dev-master" correspond à un numéro de version exact : le dernier disponible. C'est un cas particulier.

Pour information, vous pouvez aller regarder les informations de cette librairie sur [packagist](#). Vous pouvez voir qu'elle dépend d'une autre librairie, "twig/twig", qui correspond au moteur de template Twig à proprement parler. Il en a besoin dans sa version "1.*", Composer ira donc chercher la dernière version dans la branche 1.*.

Mettre à jour les dépendances

Pour mettre à jour toutes les dépendances, "twig/extensions" dans notre cas, il faut exécuter la commande `update` de Composer, comme ceci :

Code : Console

```
C:\wamp\www\test>php ./composer.phar update
Loading composer repositories with package information
Updating dependencies
- Installing twig/twig (v1.10.0)
  Downloading: 100%
- Installing twig/extensions (dev-master dcdf02)
  Cloning dcdf02fbac1282e6b8f4d0558cc7e9580105688

Writing lock file
Generating autoload files
```



Chez moi, j'ai placé Composer (le fichier `composer.phar`) dans le répertoire `www`. Or ici on travaille dans le répertoire de test `www\test`. J'ai donc dit à PHP d'exécuter le fichier `./composer.phar`, mais bien sûr si le vôtre est dans le répertoire courant ou ailleurs, adaptez la commande. 😊

Et voilà !

Vous pouvez aller vérifier dans le répertoire `test/vendor` :

- Composer a téléchargé la dépendance "twig/extensions" que l'on a défini, dans `vendor/twig/extensions` ;
- Composer a téléchargé la dépendance "twig/twig" de notre dépendance à nous, dans `vendor/twig/twig` ;
- Composer a généré les fichiers nécessaires pour l'`autoload`, allez vérifier le fichier `vendor/composer/autoload_namespaces.php`.

Tout est maintenant OK pour se servir de twig/extensions dans votre projet ! Voici donc la démarche et le fonctionnement pour la gestion des dépendances avec Composer. Mais revenons maintenant à notre projet sous Symfony2.

Mettons à jour Symfony2



Ce paragraphe s'adresse à ceux qui ont téléchargé l'archive « with vendors » de Symfony2, et qui ne l'ont pas déjà téléchargé via Composer.

Si vous avez téléchargé la version de Symfony2 qui comprend déjà les dépendances, vous avez en fait téléchargé tout le contenu du dossier `vendor` que Composer pourrait gérer tout seul. L'objectif de ce paragraphe est de déléguer cette gestion à Composer.

Vous pouvez voir qu'en fait il existe déjà un fichier de définition des dépendances à la racine de votre projet : le fichier `composer.json`. N'hésitez pas à l'ouvrir : vous pourrez y voir toutes les dépendances déjà définies.

Pour l'instant, ce fichier existe mais on n'a jamais utilisé Composer pour les gérer. Il ne reste donc plus qu'à dire à Composer de les mettre toutes à jour. Rien de spécial à faire par rapport à tout à l'heure, exécutez simplement la commande suivante :

Code : Console

```
php ../composer.phar update
```

Cela va prendre un peu de temps, car Composer a beaucoup à télécharger, les dépendances d'un projet Symfony2 sont nombreuses. Il y a en effet Symfony2 en lui-même, mais également Doctrine, Twig, certains bundles, etc.

Maintenant Composer a pris la main sur toutes vos dépendances, on va pouvoir en ajouter une nouvelle : un bundle Symfony2 !

Installer un bundle avec Composer

Dans ce chapitre nous allons installer le bundle `DoctrineFixtureBundle`, qui permet de pré-remplir la base de données avec des données, afin de bien tester votre application. Cependant, les explications sont valables pour l'installation de n'importe quel bundle, retenez donc bien la méthode.

1. Trouver le nom du bundle

Vous l'avez précédemment, on définit une dépendance dans Composer via son nom. Il faut donc logiquement connaître ce nom pour pouvoir l'installer. Pour cela rien de plus simple, on se rend sur <http://packagist.org/> et on fait une petite recherche. Dans notre cas, recherchez « fixture », et cliquez sur le bundle de Doctrine, « doctrine/doctrine-fixtures-bundle ».

2. Déterminer la version du bundle

Une fois que vous avez trouvé votre bundle, il faut en sélectionner une version. Il se peut que celui-ci n'ait pas vraiment de version fixe, et que seul « dev-master » soit disponible. Dans ce cas assurez-vous (auprès du développeur, ou en regardant le code) qu'il est compatible avec votre projet.

Mais la plupart du temps, les bundles sont versionnés et c'est à vous de choisir la version qui vous convient. Restons sur notre cas du bundle `fixture` : <https://packagist.org/packages/doctrine/...xtures-bundle>. Les deux dernières versions sont "dev-master" et "2.0.x-dev" :

- Regardez les prérequis de la version 2.0.x-dev : il est indiqué que cette version a besoin de "symfony/symfony" dans sa version 2.0 (car "<2.1" exclut la version 2.1). Cette version est trop vieille, on ne peut donc pas l'utiliser car on tourne sur un Symfony2.1 ;
- Regardez alors les prérequis de la version dev-master : on a besoin de "doctrine/doctrine-bundle" (cliquez dessus), qui lui-même a besoin de "symfony/framework-bundle" dans sa version 2.1 ou 2.2 ("<2.3" exclut la 2.3 rappelvez-vous). Ce bundle suit la même numérotation de version que le framework Symfony2, la version 2.1 est donc OK pour nous.

On choisit alors la version "dev-master" du bundle.

3. Déclarer le bundle à Composer

Une fois qu'on a le nom du bundle et sa version, il faut le déclarer à Composer, dans le fichier `composer.json`. On sait déjà le faire, il faut modifier la section "require", voici ce que ça donne :

Code : JavaScript

```
// composer.json
// ...
"require": {
    "php": ">=5.3.3",
    // ...
    "jms/di-extra-bundle": "1.1.*",
    "doctrine/doctrine-fixtures-bundle": "dev-master"
},
// ...
```



N'oubliez pas de rajouter une virgule à la fin de l'avant-dernière dépendance, dans mon cas "jms/di-extra-bundle" !

4. Mettre à jour les dépendances

Une fois la dépendance déclarée à Composer, il ne reste qu'à mettre à jour les dépendances, avec la commande `update` :

Code : Console

```
C:\wamp\www\Symfony>php ../composer.phar update
Loading composer repositories with package information
Updating dependencies
  - Installing doctrine/doctrine-fixtures (dev-master a95d783)
    Cloning a95d7839a7794c7c9b22d64e859ee70658d977fe

  - Installing doctrine/doctrine-fixtures-bundle (dev-master 9edc67a)
    Cloning 9edc67af16e736a31605e7fa9c9e3edbd9db6427

Writing lock file
Generating autoload files
Clearing the cache for the dev environment with debug true
Installing assets using the hard copy option
[...]
```

5. Enregistrer le bundle dans le Kernel

Dernier point, totalement indépendant de Composer : il faut déclarer le bundle dans le Kernel de Symfony2. Allez dans `app/AppKernel.php` et rajoutez la ligne 8 :

Code : PHP

```
<?php
// app/AppKernel.php

// ...

if (in_array($this->getEnvironment(), array('dev', 'test'))) {
    // ...
    $bundles[] = new
        Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle();
}

// ...
```



Ici j'ai déclaré le bundle uniquement pour les modes « dev » et « test », car c'est l'utilité du bundle fixture on en reparlera. Bien entendu si votre bundle doit être accessible en mode « prod », placez-le en conséquence. 😊

Voilà, votre bundle est opérationnel !

Attention si vous avez l'habitude de Symfony2.0 où il fallait également déclarer le namespace pour l'autoload : perdez cette habitude tout de suite ! Composer s'occupe vraiment de tout, et notamment de déclarer les namespaces pour l'autoload : allez le vérifier dans le fichier `vendor/composer/autoload_namespaces.php`. Ce fichier contient tous les namespaces nécessaires pour votre projet, et Symfony2 l'inclut déjà, vérifiez-le en regardant le fichier `app/autoload.php`, on y voit cette ligne :

Code : PHP

```
<?php
if (!$loader = @include __DIR__ . '/../vendor/autoload.php') {
    // ...
```

Voilà comment Symfony2 utilise Composer pour gérer son autoload.

Gérer l'autoload d'une bibliothèque manuellement

Il se peut que vous ayez une bibliothèque existante en stock, mais qui n'est pas référencée sur `packagist.org`. Composer ne peut pas gérer entièrement cette bibliothèque car il n'a pas ses informations : comment la mettre à jour, quelles sont ses dépendances, etc.

Par contre, vous avez tout de même envie de l'intégrer dans votre projet. Notamment, vous souhaitez la charger automatiquement grâce à l'`autoload` PHP. Pour ce faire, il faut rajouter la section "autoload" à votre `composer.json`, dans laquelle Composer ne mettra pas son nez pour tout ce qui est installation et mises à jour. Par contre, il l'inclura dans son fichier d'autoload que Symfony2 charge. Voici ce que vous devez rajouter :

Code : JavaScript

```
// composer.json
{
    // ...
    "autoload": {
        "psr-0": {
            "VotreNamespace": "chemin/vers/la/bibliotheque"
        }
    }
    // ...
}
```



Attention, il faut toujours utiliser cette méthode et ne jamais aller modifier le fichier `vendor/composer/autoload_namespaces.php` ! Comme tout fichier qui se trouve dans le répertoire `vendor`, vous ne devez pas le toucher car il peut être écrasé à tout moment.

Bien sûr, pour que cela fonctionne il faut que votre bibliothèque respecte la convention [PSR-0](#), c'est-à-dire une convention de nommage et d'*autoload*. Je vous invite à lire le lien pour en savoir plus dessus. Symfony2 suit bien entendu cette convention.

Pour conclure

Ce chapitre-parentthèse sur Composer touche à sa fin. S'il vous semble un peu décalé aujourd'hui, vous me remercierez de vous en avoir parlé un peu plus tard lorsque vous voudrez installer des bundles trouvés à droite ou à gauche. D'ailleurs on a déjà installé DoctrineFixtureBundle, un bundle bien pratique dont nous nous resservirons dès la prochaine partie sur Doctrine !

Sachez également que je n'ai absolument pas tout dit sur Composer, car ça ferait trop long et ce n'est pas tellement l'objet de ce tutoriel. Cependant, Composer a sa propre documentation et je vous invite à vous y référer. Par curiosité, par intérêt, par problème, n'hésitez pas : <http://getcomposer.org/> !

En résumé

- Composer est un outil pour gérer les dépendances d'un projet en PHP, qu'il soit sous Symfony2 ou non ;
- Le fichier `composer.json` permet de lister les dépendances que doit inclure Composer dans votre projet ;
- Composer détermine la meilleure version possible pour vos dépendances, les télécharge, et configure leur *autoload* tout seul ;
- Composer trouve toutes les bibliothèques sur le site <http://www.packagist.org/>, sur lequel vous pouvez envoyer votre propre bibliothèque si vous le souhaitez ;
- La très grande majorité des bundles Symfony2 sont installables avec Composer, ce qui simplifie énormément leur utilisation dans un projet.

Partie 3 : Gérer la base de données avec Doctrine2

Symfony2 est livré par défaut avec l'ORM Doctrine2. Qu'est-ce qu'un ORM ? Qu'est-ce que Doctrine2 ? Ce tutoriel pour débutants est fait pour vous, car c'est ce que nous allons apprendre dans cette partie !

Lisez bien l'ensemble des chapitres de cette partie : ils forment un tout, et toutes vos questions seront résolues à la fin de la partie !

La couche métier : les entités

L'objectif d'un ORM (pour *Object-Relation Mapper*, soit en français « Lien Objet-Relation ») est simple : se charger de l'enregistrement de vos données en vous faisant oublier que vous avez une base de données. Comment ? En s'occupant de tout ! Nous n'allons plus écrire de requêtes, ni créer de tables *via* phpMyAdmin. Dans notre code PHP, nous allons faire appel à Doctrine2, l'ORM par défaut de Symfony2, pour faire tout cela.

Notions d'ORM : fini les requêtes, utilisons des objets

Je vous propose de commencer par un exemple pour bien comprendre. Supposons que vous disposez d'une variable `<?php $utilisateur`, un objet `User` qui représente l'un de vos utilisateurs qui vient de s'inscrire sur votre site. Pour sauvegarder cet objet, vous êtes habitué à créer votre propre fonction qui effectue une requête SQL du type `INSERT INTO` dans la bonne table, etc. Bref, vous devez gérer tout ce qui touche à l'enregistrement en base de données. En utilisant un ORM, vous n'aurez plus qu'à utiliser quelques fonctions de cet ORM, par exemple : `<?php $orm->save($utilisateur)`. Et ce dernier s'occupera de tout ! Vous avez enregistré votre utilisateur en une seule ligne 😊. Bien sûr, ça n'est qu'un exemple, nous verrons les détails pratiques dans la suite de ce chapitre, mais retenez bien l'idée.

Mais l'effort que vous devrez faire pour bien utiliser un ORM, c'est d'oublier votre côté « administrateur de base de données ». Oubliez les requêtes SQL, pensez objet !

Vos données sont des objets

Dans ORM, il y a la lettre O comme `Objet`. En effet, pour que tout le monde se comprenne, toutes vos données doivent être sous forme d'objets. Concrètement, qu'est-ce que cela implique dans notre code ? Pour reprendre l'exemple de notre utilisateur, quand vous étiez petit, vous utilisiez sûrement un tableau puis vous accédiez à vos attributs *via* `<?php $utilisateur['pseudo']` ou `<?php $utilisateur['email']` par exemple. Soit, c'était très courageux de votre part. Mais nous allons aller plus loin, maintenant.

Utiliser des objets n'est pas une grande révolution en soi. Faire `<?php $utilisateur->getPseudo()` au lieu de `<?php $utilisateur['pseudo']`, c'est joli, mais limité. Ce qui est une révolution, c'est de coupler cette représentation objet avec l'ORM. Qu'est-ce que vous pensez d'un `<?php $utilisateur->getCommentaires()` ? Ahah ! Vous ne pouvez pas faire cela avec votre tableau ! Ici, la méthode `<?php $utilisateur->getCommentaires()` déclencherait la bonne requête, récupérera tous les commentaires postés par votre utilisateur, et vous retournerait une sorte de tableau d'objets de type `Commentaire` que vous pourriez afficher sur la page de profil de votre utilisateur, par exemple. Ça commence à devenir intéressant, n'est-ce pas ?

Au niveau du vocabulaire, un objet dont vous confiez l'enregistrement à l'ORM s'appelle une `entité` (*entity* en anglais). On dit également `persister` une entité, plutôt qu'enregistrer une entité. Vous savez, l'informatique et le jargon... 😊

Créer une première entité avec Doctrine2

Une entité, c'est juste un objet

Derrière ce titre se cache la vérité. Une entité, ce que l'ORM va manipuler et enregistrer dans la base de données, ce n'est vraiment rien d'autre qu'un simple objet. Voici ce à quoi pourrait ressembler l'objet `Article` de notre blog :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

class Article
{
    protected $id;
    protected $date;
    protected $titre;
    protected $auteur;
    protected $contenu;

    // Et bien sûr les getter/setter :

    public function setId($id)
    {
        $this->id = $id;
    }
    public function getId()
    {
        return $this->id;
    }
}
```

```

public function setDate($date)
{
    $this->date = $date;
}
public function getDate()
{
    return $this->date;
}

public function setTitre($titre)
{
    $this->titre = $titre;
}
public function getTitre()
{
    return $this->titre;
}

public function setAuteur($auteur)
{
    $this->auteur = $auteur;
}
public function getAuteur()
{
    return $this->auteur;
}

public function setContenu($contenu)
{
    $this->contenu = $contenu;
}
public function getContenu()
{
    return $this->contenu;
}
}

```



Inutile de créer ce fichier pour l'instant, nous allons le générer plus bas, patience. 😊

Comme vous pouvez le voir, c'est très simple. Un objet, des propriétés, et bien sûr, les *getters/setters* correspondants. On pourrait en réalité utiliser notre objet dès maintenant !

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Sdz\BlogBundle\Entity\Article;

// ...

public function testAction()
{
    $article = new Article();
    $article->setDate(new \Datetime()); // date d'aujourd'hui
    $article->setTitre('Mon dernier weekend');
    $article->setAuteur('Bibi');
    $article->setContenu("C'était vraiment super et on s'est bien
amusé.");
}

return $this->render('SdzBlogBundle:Article:test.html.twig',
array('article' => $article));
}

```

Rajoutez à cela la vue correspondante qui afficherait l'article passé en argument avec un joli code HTML, et vous avez un code opérationnel. Bien sûr il est un peu limité car statique, mais l'idée est là et vous voyez comment l'on peut se servir d'une entité. Retenez donc : une entité n'est rien d'autre qu'un objet.

Normalement, vous devez vous poser une question : comment l'ORM va-t-il faire pour enregistrer cet objet dans la base de données s'il ne connaît rien de nos propriétés `date`, `titre` et `contenu`? Comment peut-il deviner que notre propriété `date` doit être stockée avec un champ de type `DATE` dans la table ? La réponse est aussi simple que logique : il ne devine rien, on va le lui dire !

Une entité, c'est juste un objet... mais avec des commentaires !



Quoi ? Des commentaires ?

OK, je dois avouer que ça n'est pas intuitif si vous ne vous en êtes jamais servi, mais oui, on va rajouter des commentaires dans notre code et Symfony2 va se servir directement de ces commentaires pour ajouter des fonctionnalités à notre application. Ce type de commentaire se nomme **l'annotation**. Les annotations doivent respecter une syntaxe particulière, regardez par vous-même :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

// On définit le namespace des annotations utilisées par Doctrine2
// En effet il existe d'autres annotations, on le verra par la suite,
// qui utiliseront un autre namespace
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
@ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
*/
class Article
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(name="date", type="date")
     */
    private $date;

    /**
     * @ORM\Column(name="titre", type="string", length=255)
     */
    private $titre;

    /**
     * @ORM\Column(name="auteur", type="string", length=255)
     */
    private $auteur;

    /**
     * @ORM\Column(name="contenu", type="text")
     */
    private $contenu;

    // Les getters
    // Les setters
}
```



Ne recopiez toujours pas toutes ces annotations à la main, on utilise le générateur en console au paragraphe juste en dessous.



Attention par contre pour les prochaines annotations que vous serez amené à écrire à la main : elles doivent être dans des commentaires de type « `/**` », avec précisément deux étoiles. Si vous essayez de les mettre dans un commentaire de type « `/* */` ou encore « `//` », elles seront simplement ignorées.

Grâce à ces annotations, Doctrine2 dispose de toutes les informations nécessaires pour utiliser notre objet, créer la table correspondante, l'enregistrer, définir un identifiant (id) en auto-incrémentation, nommer les colonnes, etc. Ces informations se nomment les *metadata* de notre entité. Je ne vais pas épiloguer sur les annotations, elles sont suffisamment claires pour être comprises par tous. 😊 Ce qu'on vient de faire, à savoir rajouter les *metadata* à notre objet Article s'appelle *mapper* l'objet Article. C'est-à-dire faire le lien entre notre objet de base et la représentation physique qu'utilise Doctrine2.

Sachez quand même que, bien que l'on utilisera les annotations tout au long de ce tutoriel, il existe d'autres moyens de définir les *metadata* d'une entité : en YAML, en XML et en PHP. Si cela vous intéresse, vous trouverez plus d'informations sur la définition des *metadata* via les autres moyens dans le chapitre [Doctrine2 de la documentation de Symfony2](#).

Générer une entité : le générateur à la rescousse !

En tant que bon développeur, on est fainéant à souhait, et ça, Symfony2 l'a bien compris ! On va donc se refaire une petite session en console afin de générer notre première entité. Entrez la commande suivante et suivez le guide :

Code : Console

```
C:\wamp\www\Symfony>php app/console generate:doctrine:entity
```

1. **Code : Console**

```
Welcome to the Doctrine2 entity generator
```

```
This command helps you generate Doctrine2 entities.
```

```
First, you need to give the entity name you want to generate.  
You must use the shortcut notation like AcmeBlogBundle:Post.
```

```
The Entity shortcut name:_
```

Grâce à ce que le générateur vous dit, vous l'avez compris, il faut rentrer le nom de l'entité sous le format NomBundle:NomEntité. Dans notre cas, on entre donc `SdzBlogBundle:Article` ;

2.

Code : Console

```
The Entity shortcut name: SdzBlogBundle:Article
```

```
Determine the format to use for the mapping information.
```

```
Configuration format (yml, xml, php, or annotation) [annotation]:_
```

Comme je vous l'ai dit, nous allons utiliser les annotations, qui sont d'ailleurs le format par défaut. Appuyez juste sur la touche Entrée ;

3.

Code : Console

```
Configuration format (yml, xml, php, or annotation) [annotation]:
```

```
Instead of starting with a blank entity, you can add some fields now.  
Note that the primary key will be added automatically (named id).
```

```
Available types: array, object, boolean, integer, smallint,  
bigint, string, text, datetime, datetimetz, date, time, decimal, float,  
blob.
```

```
New field name (press <return> to stop adding fields):_
```

On commence à saisir le nom de nos champs. Lisez bien ce qui est inscrit avant : Doctrine2 va ajouter automatiquement l'id, de ce fait, pas besoin de le redéfinir ici. On entre donc notre date : `date` ;

4.

Code : Console

```
New field name (press <return> to stop adding fields): date  
Field type [string]:_
```

C'est maintenant que l'on va dire à Doctrine à quel type correspond notre propriété date. Voici la liste des types possibles :
`array, object, boolean, integer, smallint, bigint, string, text, datetime, datetimetz, date, time, decimal, et float.` Tapez donc `datetime` ;

5. Répétez les points 3 et 4 pour les propriétés titre, auteur et contenu. Titre et Auteur sont de type `string` de 255 caractères (pourquoi pas). Contenu est par contre de type `text` ;

6.

Code : Console

```
New field name (press <return> to stop adding fields): date
Field type [string]: datetime

New field name (press <return> to stop adding fields): titre
Field type [string]: string
Field length [255]: 255

New field name (press <return> to stop adding fields): auteur
Field type [string]: string
Field length [255]: 255

New field name (press <return> to stop adding fields): contenu
Field type [string]: text

New field name (press <return> to stop adding fields):_
```

Lorsque vous avez fini, appuyez sur la touche Entrée ;
7.

Code : Console

```
New field name (press <return> to stop adding fields):
Do you want to generate an empty repository class [no]?_
```

Oui, on va créer le *repository* associé, c'est très pratique, nous en reparlerons largement. Entrez donc **yes** ;
8. Confirmez la génération, et voilà !

Code : Console

```
Do you want to generate an empty repository class [no]? yes
Summary before generation

You are going to generate a "SdzBlogBundle:Article" Doctrine2 entity
using the "annotation" format.

Do you confirm generation [yes]?

Entity generation

Generating the entity code: OK

You can now start using the generated code!

C:\wamp\www\Symfony>_
```

Allez tout de suite voir le résultat dans le fichier `Entity/Article.php`. Symfony2 a tout généré, même les *getters* et les *setters* ! Vous êtes l'heureux propriétaire d'une simple classe... avec plein d'annotations !



On a utilisé le générateur de code pour nous faciliter la vie. Mais sachez que vous pouvez tout à fait vous en passer ! Comme vous pouvez le voir, le code généré n'est pas franchement compliqué, et vous pouvez bien entendu l'écrire à la main si vous préférez.

Affiner notre entité avec de la logique métier

L'exemple de notre entité `Article` est un peu simple, mais rappelez-vous que la couche modèle dans une application est la couche métier. C'est-à-dire qu'en plus de gérer vos données, un modèle contient également la logique de l'application. Voyez par vous-même avec les exemples ci-dessous.

Attributs calculés

Prenons l'exemple d'une entité `Commande`, qui représenterait un ensemble de produits à acheter sur un site d'e-commerce. Cette entité aurait les attributs suivants :

- `ListeProduits` qui contient un tableau des produits de la commande ;
- `AdresseLivraison` qui contient l'adresse où expédier la commande ;
- `Date` qui contient la date de la prise de la commande ;
- Etc.

Ces trois attributs devront bien entendu être *mappés* (c'est-à-dire défini comme des colonnes pour l'ORM *via* des annotations) pour être enregistrés en base de données par Doctrine2. Mais il existe d'autres caractéristiques pour une commande, qui nécessitent un peu de calcul : le prix total, un éventuel coupon de réduction, etc. Ces caractéristiques n'ont pas à être persistées en base de données, car elles peuvent être déduites des informations que l'on a déjà. Par exemple pour avoir le prix total, il suffit de faire une boucle sur `ListeProduits` et d'additionner les prix de chaque produit :

Code : PHP

```
<?php
// Exemple :
class Commande
{
    public function getPrixTotal()
    {
        $prix = 0;
        foreach($this->getListeProduits() as $produit)
        {
            $prix += $produit->getPrix();
        }
        return $prix;
    }
}
```

N'hésitez donc pas à créer des méthodes `getQuelquechose()` qui contiennent de la logique métier. L'avantage de mettre la logique dans l'entité même est que vous êtes sûr de réutiliser cette même logique partout dans votre application. Il est bien plus propre et pratique de faire `<?php $commande->getPrixTotal()` que d'éparpiller à droite et à gauche différentes manières de calculer ce prix total. Bien sûr, ces méthodes n'ont pas d'équivalent `setQuelquechose()`, cela n'a pas de sens !

Attributs par défaut

Vous avez aussi parfois besoin de définir une certaine valeur à vos entités lors de leur création. Or nos entités sont de simples objets PHP, et la création d'un objet PHP fait appel... au constructeur. Pour notre entité `Article` on pourrait définir le constructeur suivant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Article
{
    // La définition des attributs...

    public function __construct()
    {
        $this->date = new \Datetime(); // Par défaut, la date de l'article est la date d'aujourd'hui
    }

    // Les getter/setter...
}
```

Conclusion

N'oubliez pas : une entité est un objet PHP qui correspond à un besoin dans votre application.

N'essayez donc pas de raisonner en termes de tables, base de données, etc. Vous travaillez maintenant avec des objets PHP, qui contiennent une partie de logique métier, et qui peuvent se manipuler facilement. C'est vraiment important que vous fassiez l'effort dès maintenant de prendre l'habitude de manipuler des objets, et non des tables.

Tout sur le mapping !

Vous avez rapidement vu comment *mapper* vos objets avec les annotations. Mais ces annotations permettent d'inscrire pas mal d'autres informations. Il faut juste en connaître la syntaxe, c'est l'objectif de cette section.

Tout ce qui va être décrit ici se trouve bien entendu dans [la documentation officielle sur le mapping](#), que vous pouvez garder à

portée de main.

L'annotation Entity

L'annotation Entity s'applique sur une classe, il faut donc la placer avant la définition de la classe en PHP. Elle définit un objet comme étant une entité, et donc persisté par Doctrine. Cette annotation s'écrit comme suit :

Code : Autre

```
@ORM\Entity
```

Il existe un seul paramètre facultatif pour cette annotation, `repositoryClass`. Il permet de préciser le namespace complet du `repository` qui gère cette entité. Nous donnerons le même nom à nos `repository` qu'à nos entités, en les suffisant simplement de « `Repository` ». Pour notre entité `Article`, cela donne :

Code : Autre

```
@ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
```



Un `repository` sert à récupérer vos entités depuis la BDD, on en reparle dans un chapitre dédié plus loin dans le cours.

L'annotation Table

L'annotation Table s'applique sur une classe également. C'est une annotation facultative, une entité se définit juste par son annotation Entity. Cependant l'annotation Table permet de personnaliser le nom de la table qui sera créée dans la base de données. Par exemple, on pourrait préfixer notre table article par « `sdz` » :

Code : Autre

```
@ORM\Table(name="sdz_article")
```

Elle se positionne juste avant la définition de la classe.



Par défaut, si vous ne précisez pas cette annotation, le nom de la table créée par Doctrine2 est le même que celui de l'entité. Dans notre cas, cela aurait été « `Article` » (avec la majuscule donc, attention).

L'annotation Column

L'annotation Column s'applique sur un attribut de classe, elle se positionne donc juste avant la définition PHP de l'attribut concerné. Cette annotation permet de définir les caractéristiques de la colonne concernée. Elle s'écrit comme suit :

Code : Autre

```
@ORM\Column
```

L'annotation Column comprend quelques paramètres, dont le plus important est le type de la colonne.

Les types de colonnes

Les types de colonnes que vous pouvez définir en annotation sont des types Doctrine, *et uniquement Doctrine*. Ne les confondez pas avec leurs homologues SQL ou PHP, ce sont des types à Doctrine seul. Ils font la transition des types SQL aux types PHP.

Voici dans le tableau suivant la liste exhaustive des types Doctrine2 disponibles.

Type Doctrine	Type SQL	Type PHP	Utilisation
string	VARCHAR	string	Toutes les chaînes de caractères jusqu'à 255 caractères.
integer	INT	integer	Tous les nombres jusqu'à 2 147 483 647.
smallint	SMALLINT	integer	Tous les nombres jusqu'à 32 767.

bigint	BIGINT	string	Tous les nombres jusqu'à 9 223 372 036 854 775 807. Attention PHP reçoit une chaîne de caractères car il ne supporte pas un si grand nombre (suivant que vous êtes en 32 ou en 64 bits).
boolean	BOOLEAN	boolean	Les valeurs booléennes true et false.
decimal	DECIMAL	double	Les nombres à virgules.
date ou datetime	DATETIME	objet DateTime	Toutes les dates et heures.
time	TIME	objet DateTime-	Toutes les heures.
text	CLOB	string	Les chaînes de caractères de plus de 255 caractères.
object	CLOB	Type de l'objet stocké	Stocke un objet PHP en utilisant serialize/unserialize.
array	CLOB	array	Stocke un tableau PHP en utilisant serialize/unserialize.
float	FLOAT	double	Tous les nombres à virgules. Attention, fonctionne uniquement sur les serveurs dont la locale utilise un point comme séparateur.



Les types Doctrine sont sensibles à la casse. Ainsi, le type « String » n'existe pas, il s'agit du type « string ». Facile à retenir : tout est en minuscule !

Le type de colonne se définit en tant que paramètre de l'annotation Column, comme suit :

Code : Autre

```
@ORM\Column(type="string")
```

Les paramètres de l'annotation Column

Il existe 7 paramètres, tous facultatifs, que l'on peut passer à l'annotation Column afin de personnaliser le comportement. Voici la liste exhaustive dans le tableau suivant.

Paramètre	Valeur par défaut	Utilisation
type	string	Définit le type de colonne comme nous venons de voir.
name	Nom de l'attribut	Définit le nom de la colonne dans la table. Par défaut, le nom de la colonne est le nom de l'attribut de l'objet, ce qui convient parfaitement. Mais vous pouvez changer le nom de la colonne, par exemple si vous préférez « isExpired » en attribut mais « is_expired » dans la table.
length	255	Définit la longueur de la colonne. Applicable uniquement sur un type de colonne <i>string</i> .
unique	false	Définit la colonne comme unique. Par exemple sur une colonne email pour vos membres.
nullable	false	Permet à la colonne de contenir des NULL.
precision	0	Définit la précision d'un nombre à virgule, c'est-à-dire le nombre de chiffres en tout. Applicable uniquement sur un type de colonne <i>decimal</i> .
scale	0	Définit le scale d'un nombre à virgule, c'est-à-dire le nombre de chiffres après la virgule. Applicable uniquement sur un type de colonne <i>decimal</i> .

Pour définir plusieurs options en même temps, il faut simplement les séparer avec une virgule. Par exemple, pour une colonne « email » en string 255 et unique il faudra faire :

Code : Autre

```
@ORM\Column(type="string", length=255, unique=true)
```

Pour conclure

Vous savez maintenant tout ce qu'il faut savoir sur la couche Modèle sous Symfony2 en utilisant les entités de l'ORM Doctrine2.

Je vous redonne l'adresse de la documentation Doctrine2, que vous serez amené à utiliser maintes fois dans votre

développement : <http://docs.doctrine-project.org/projet/index.html>. J'insiste : enregistrez-la dans vos favoris car Doctrine est une bibliothèque très large, et si je vais vous donner un maximum d'information dans cette partie du tutoriel, je ne pourrai pas tout couvrir.



Attention, Doctrine étant une bibliothèque totalement indépendante de Symfony2, sa documentation fait référence à ce type d'annotation : `/** @Entity */`
Il faut impérativement l'adapter à votre projet Symfony2, en préfixant toutes les annotations par "ORM\" comme nous l'avons vu dans ce chapitre : `/** @ORM\Entity */`
Car dans nos entités, c'est le namespace ORM que nous chargeons. Ainsi, l'annotation `@Entity` n'existe pas pour nous, c'est `@ORM` qui existe (et tous ses enfants : `@ORM\Entity`, `@ORM\Table`, etc.).

Dans le prochain chapitre, nous apprendrons à manipuler les entités que nous savons maintenant construire.

En résumé

- Le rôle d'un ORM est de se charger de la persistance de vos données : vous manipulez des objets, et lui s'occupe de les enregistrer en base de données ;
- L'ORM par défaut livré avec Symfony2 est Doctrine2 ;
- L'utilisation d'un ORM implique un changement de raisonnement : on utilise des objets, et on raisonne en POO. C'est au développeur de s'adapter à Doctrine2 et non l'inverse !
- Une entité est, du point de vue PHP, un simple objet. Du point de vue de Doctrine, c'est un objet complété avec des informations de *mapping* qui lui permettent d'enregistrer correctement l'objet en base de données ;
- Une entité est, du point de vue de votre code, un objet PHP qui correspond à un besoin, et indépendant du reste de votre application.

Manipuler ses entités avec Doctrine2

Dans ce chapitre, nous allons apprendre à manipuler les entités que vous avez faites au précédent chapitre.

Mérialiser les tables en base de données

Avant de pouvoir utiliser notre entité comme il se doit, on doit d'abord créer la table correspondante dans la base de données !

Créer la table correspondante dans la base de données

Alors, j'espère que vous avez installé et configuré phpMyAdmin, on va faire de la requête SQL !

...

Ceux qui m'ont cru, relisez le chapitre précédent. 😊 Les autres, venez, on est bien trop fainéants pour ouvrir phpMyAdmin !

Avant toute chose, vérifiez que vous avez bien configuré l'accès à votre base de données dans Symfony2. Si ce n'est pas le cas, il suffit d'ouvrir le fichier `app/config/parameters.yml` et de mettre les bonnes valeurs aux lignes commençant par `database_`: serveur, nom de la base, nom d'utilisateur et mot de passe. Vous avez l'habitude de ces paramètres :

Code : YAML

```
# app/config/parameters.yml

parameters:
    database_driver: pdo_mysql
    database_host: localhost
    database_port: ~
    database_name: symfony
    database_user: root
    database_password: ~
```

Ensuite, direction la console. Cette fois-ci, on ne va pas utiliser une commande du `generator` mais une commande de **Doctrine**, car on ne veut pas générer du code mais une table dans la base de données.

D'abord, si vous ne l'avez pas déjà fait, il faut créer la base de données. Pour cela, exécutez la commande (vous n'avez à le faire qu'une seule fois évidemment) :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:database:create
Created database for connection named `symfony`

C:\wamp\www\Symfony>_
```

Ensuite, il faut générer les tables à l'intérieur de cette base de données. Exécutez donc la commande suivante :

Code : Console

```
php app/console doctrine:schema:update --dump-sql
```

Cette dernière commande est vraiment performante. Elle va comparer l'état actuel de la base de données avec ce qu'elle devrait être en tenant compte de toutes nos entités. Puis, elle affiche les requêtes SQL à exécuter pour passer de l'état actuel au nouvel état.

En l'occurrence, nous avons seulement créé une entité, donc la différence entre l'état actuel (base de données vide) et le nouvel état (base de données avec une table article) n'est que d'une seule requête SQL : la requête de création de la table. Doctrine vous affiche donc cette requête :

Code : SQL

```
CREATE TABLE Article (id INT AUTO_INCREMENT NOT NULL,
date DATETIME NOT NULL,
titre VARCHAR(255) NOT NULL,
auteur VARCHAR(255) NOT NULL,
contenu LONGTEXT NOT NULL,
PRIMARY KEY(id)) ENGINE = InnoDB;
```

Pour l'instant, rien n'a été fait en base de données, Doctrine nous a seulement affiché la ou les requêtes qu'il s'apprête à exécuter. Pensez à toujours valider rapidement ces requêtes, pour être sûr de ne pas avoir fait d'erreur dans le *mapping* des entités. Mais maintenant, il est temps de passer aux choses sérieuses, et d'exécuter concrètement cette requête ! Lancez la commande suivante :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:schema:update --force
Updating database schema...
Database schema updated successfully! "1" queries were executed
C:\wamp\www\Symfony>_
```

Si tout se passe bien, vous avez le droit au `Database schema updated successfully!`. Génial, mais bon, vérifions-le quand même. Cette fois-ci, ouvrez phpMyAdmin (vraiment, ça n'est pas un piège), allez dans votre base de données et voyez le résultat : la table `article` a bien été créée avec les bonnes colonnes, l'id en auto-incrémentation, etc. C'est super !

Modifier une entité

Pour modifier une entité, il suffit de lui créer un attribut et de lui attacher l'annotation correspondante. Faisons-le dès maintenant en rajoutant un attribut `$publication`, un booléen qui indique si l'article est publié (`true` pour l'afficher sur la page d'accueil, `false` sinon), ce n'est qu'un exemple bien entendu. Rajoutez donc ces lignes dans votre entité :

Code : PHP

```
<?php
// sdz/Sdz/BlogBundle/Entity/Article.php

class Article
{
    // ...

    /**
     * @ORM\Column(name="publication", type="boolean")
     */
    private $publication;

    // Et modifions le constructeur pour mettre cet attribut
    // publication à true par défaut
    public function __construct()
    {
        $this->date = new \Datetime();
        $this->publication = true;
    }

    // ...
}
```

Ensuite, soit vous écrivez vous-mêmes le getter `getPublication` et le setter `setPublication`, soit vous faites comme moi et vous utilisez le générateur !

Après la commande `doctrine:generate:entity` pour générer une entité entière, vous avez la commande `doctrine:generate:entities`. C'est une commande qui génère les entités en fonction du *mapping* que Doctrine connaît. Lorsque vous faites votre *mapping* en Yaml, il peut générer toute votre entité. Dans notre cas, nous faisons notre *mapping* en annotation alors nous avons déjà défini l'attribut. La commande va donc générer ce qu'il manque : le getter et le setter !

Allons-y :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:generate:entities SdzBlogBundle:Article
Generating entity "Sdz\BlogBundle\Entity\Article"
  > backing up Article.php to Article.php~
  > generating Sdz\BlogBundle\Entity\Article
```

Allez vérifier votre entité, tout en bas de la classe le générateur a rajouté les méthodes `getPublication()` et `setPublication()`.



Vous pouvez voir également qu'il a sauvegardé l'ancienne version de votre entité dans un fichier nommé `Article.php~` : vérifiez toujours son travail, et si celui-ci ne vous convient pas, vous avez votre sauvegarde. 😊

Maintenant, il ne reste plus qu'à enregistrer ce schéma en base de données. Exécutez donc :

Code : Console

```
php app/console doctrine:schema:update --dump-sql
```

... pour vérifier que la requête est bien :

Code : SQL

```
ALTER TABLE article ADD publication TINYINT(1) NOT NULL
```

C'est le cas, cet outil de Doctrine est vraiment pratique ! Puis, exécutez la commande pour modifier effectivement la table correspondante :

Code : Console

```
php app/console doctrine:schema:update --force
```

Et voilà ! Votre entité a un nouvel attribut qui sera persisté en base de données lorsque vous l'utiliserez.

À retenir

À chaque modification du *mapping* des entités, ou lors de l'ajout/suppression d'une entité, il faudra répéter ces commandes `doctrine:schema:update --dump-sql` puis `--force` pour mettre à jour la base de données.

Enregistrer ses entités avec l'EntityManager

Maintenant, apprenons à manipuler nos entités. On va apprendre à le faire en deux parties : d'abord l'enregistrement en base de données, ensuite la récupération depuis la base de données. Mais d'abord, étudions un petit peu le service Doctrine.

Les services Doctrine2

Un service est une classe qui rempli une fonction bien précise, accessible partout dans notre code. Nous verrons cette notion de service, très importante dans Symfony2, dans un [prochain chapitre](#). Pour l'instant, concentrons nous sur ce qui nous intéresse : accéder aux fonctionnalités Doctrine2.

Le service Doctrine

Le service Doctrine est celui qui va nous permettre de gérer la persistance de nos objets. Ce service est accessible depuis le contrôleur comme n'importe quel service :

Code : PHP

```
<?php  
$doctrine = $this->get('doctrine');
```

Mais, afin de profiter de l'autocomplétion de votre IDE, la classe `Controller` de Symfony2 intègre un raccourci. Il fait exactement la même chose mais est plus joli et permet l'autocomplétion :

Code : PHP

```
<?php  
$doctrine = $this->getDoctrine();
```

C'est donc ce service Doctrine qui va nous permettre de gérer la base de données. Il permet de gérer deux choses :

- Les différentes connexions à des bases de données. C'est la partie DBAL de Doctrine2. En effet vous pouvez tout à fait utiliser plusieurs connexions à plusieurs bases de données différentes. Cela n'arrive que dans des cas particuliers, mais c'est toujours bon à savoir que Doctrine le gère bien. Le service Doctrine dispose donc, entre autres, de la méthode `$doctrine->getConnection ($name)` qui permet de récupérer une connexion à partir de son nom. Cette partie DBAL permet à Doctrine2 de fonctionner sur plusieurs types de SGBDR, tels que MySQL, PostgreSQL, etc.
- Les différents gestionnaires d'entités, ou *EntityManager*. C'est la partie ORM de Doctrine2. Encore une fois c'est logique, vous pouvez bien sûr utiliser plusieurs gestionnaires d'entités, ne serait-ce qu'un par connexion ! Le service dispose donc, entre autres, de la méthode dont nous nous servirons beaucoup : `$doctrine->getManager ($name)` qui permet de récupérer un ORM à partir de son nom.



Dans la suite du tutoriel, je considère que vous n'avez qu'un seul *EntityManager*, ce qui est le cas par défaut. La méthode `getManager ()` permet de récupérer l'*EntityManager* par défaut en omettant l'argument `$name`. J'utiliserai donc toujours `$doctrine->getManager ()` sans argument, mais pensez à adapter si ce n'est pas votre cas !



Si vous souhaitez utiliser plusieurs EntityManager, vous pouvez vous référer à la documentation officielle qui l'explique.

Le service EntityManager

On vient de le voir, le service qui va nous intéresser vraiment n'est pas doctrine, mais l'*EntityManager* de Doctrine. Vous savez déjà le récupérer depuis le contrôleur *via* :

Code : PHP

```
<?php  
$em = $this->getDoctrine()->getManager();
```

Mais sachez que, comme tout service qui se respecte, vous pouvez y accéder directement *via* :

Code : PHP

```
<?php  
$em = $this->get('doctrine.orm.entity_manager');
```

Mais attention, la première méthode vous assure l'autocomplétion alors que la deuxième non. 😊

C'est avec l'*EntityManager* que l'on va passer le plus clair de notre temps. C'est lui qui permet de dire à Doctrine « *Persiste cet objet* », c'est lui qui va exécuter les requêtes SQL (que l'on ne verra jamais), bref, c'est lui qui fera tout.

La seule chose qu'il ne sait pas faire facilement, c'est récupérer les entités depuis la base de données. Pour faciliter l'accès aux objets, on va utiliser des *Repository*.

Les services Repository

Les *Repository* sont des services qui utilisent un *EntityManager* dans les coulisses, mais qui sont bien plus faciles et pratiques à utiliser de notre point de vue. Je parle des *Repository* au pluriel car il en existe **un par entité**. Quand on parle d'un *Repository* en particulier, il faut donc toujours préciser le *Repository* de quelle entité, afin de bien savoir de quoi on parle.

On accède à ces *Repository* de la manière suivante :

Code : PHP

```
<?php  
$em = $this->getDoctrine()->getManager();  
$repository_article = $em->getRepository('SdzBlogBundle:Article');
```

L'argument de la méthode `getRepository` est l'entité pour laquelle récupérer le repository. Il y a deux manière de spécifier l'entité voulu :

- Soit en utilisant le namespace complet de l'entité. Pour notre exemple, cela donnerait :
`'Sdz\BlogBundle\Entity\Article'`.
- Soit en utilisant le raccourci `Nom_du_bundle:Nom_de_l'entité`. Pour notre exemple, c'est donc
`'SdzBlogBundle:Article'`. C'est un raccourci qui fonctionne partout dans Doctrine.

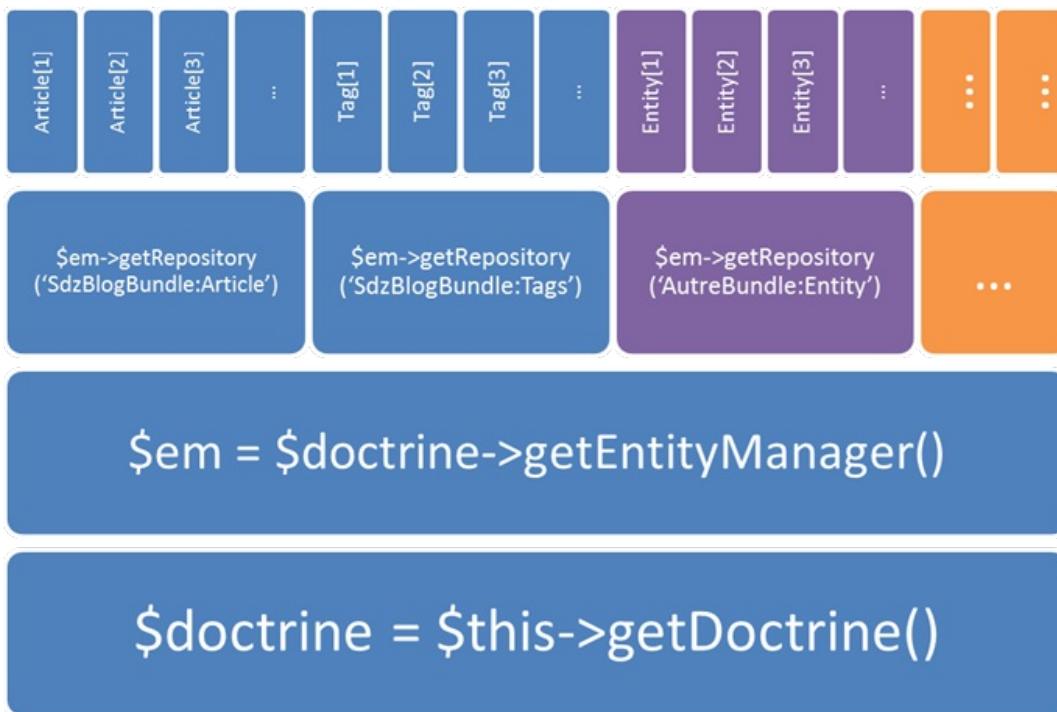


Attention, ce raccourci ne fonctionne que si vous avez mis vos entités dans le namespace `Entity` dans votre bundle.

Ce sont donc ces *repository* qui nous permettront de récupérer nos entités. Ainsi, pour charger deux entités différentes, il faut d'abord récupérer leur repository respectif. Un simple plis à prendre, mais très logique.

Conclusion

Vous savez maintenant accéder aux principaux acteurs que nous allons utiliser pour manipuler nos entités. Il reviendront très souvent, sachez les récupérer par cœur, cela vous facilitera la vie. Afin de bien les visualiser, je vous propose à la figure suivante un petit schéma à avoir en tête.



Schématisation de l'organisation de Doctrine2

Enregistrer ses entités en base de données

Rappelez-vous on a déjà vu comment créer une entité. Maintenant que l'on a cette magnifique entité entre les mains, il faut la donner à Doctrine pour qu'il l'enregistre en base de données. L'enregistrement effectif en base de données se fait en deux étapes très simples depuis un contrôleur. Modifiez la méthode `ajouterAction()` de notre contrôleur pour faire les tests :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// Attention à bien rajouter ce use en début de contrôleur
use Sdz\BlogBundle\Entity\Article;

// ...

public function ajouterAction()
{
    // Création de l'entité
    $article = new Article();
    $article->setTitre('Mon dernier weekend');
    $article->setAuteur('Bibi');
    $article->setContenu("C'était vraiment super et on s'est bien
amusé.");
    // On peut ne pas définir ni la date ni la publication
    // Car ces attributs sont définis automatiquement dans le
constructeur

    // On récupère l'EntityManager
    $em = $this->getDoctrine()->getManager();

    // Etape 1 : On « persiste » l'entité
    $em->persist($article);

    // Etape 2 : On « flush » tout ce qui a été persisté avant
    $em->flush();

    // Reste de la méthode qu'on avait déjà écrit
    if( $this->get('request')->getMethod() == 'POST' )
    {
        $this->get('session')->setFlash('notice', 'Article bien
enregistré');
        return $this->redirect( $this->generateUrl('sdzblog_voir',
array('id' => $article->getId())) );
    }

    return $this->render('SdzBlogBundle:Blog:ajouter.html.twig');
}
```

Reprenez ce code :

- La ligne 12 permet de créer l'entité, et les lignes 13 et 14 de renseigner ses attributs ;
- La ligne 18 permet de récupérer l'*EntityManager*, on en a déjà parlé je ne reviens pas dessus ;
- L'étape 1 dit à Doctrine de « persister » l'entité. Cela veut dire qu'à partir de maintenant cette entité (qui n'est qu'un simple objet !) est gérée par Doctrine. Cela n'exécute pas encore de requête SQL ni rien.
- L'étape 2 dit à Doctrine d'exécuter effectivement les requêtes nécessaires pour sauvegarder les entités qu'on lui a dit de persister ;
- Ligne 30, notre Article étant maintenant enregistré en base de données grâce au `flush()`, Doctrine2 lui a attribué un id !

Allez sur la page [/blog/ajouter](#), et voilà, vous venez d'ajouter un article dans la base de données !

Si la requête SQL effectuée vous intéresse, je vous invite à cliquer sur l'icône tout à droite dans la barre d'outil Symfony2 en bas de la page, comme le montre la figure suivante.



Vous arrivez alors dans la partie Doctrine du *profiler* de Symfony2, et vous pouvez voir les différentes requêtes SQL exécutées par Doctrine. C'est très utile pour vérifier la valeur des paramètres, la structure des requêtes, etc. N'hésitez pas à y faire des tours !

Queries

Connection default

```
+ SET NAMES UTF8
Parameters: {}
Time: 0.14 ms

INSERT INTO Article (date, titre, contenu) VALUES (?, ?, ?)
Parameters: {1: Object(DateTime), 2: 'Mon dernier weekend', 3: 'C'était vraiment super et on s'est bien amusé.'}
Time: 31.96 ms
```

Database Connections

On voit la requête

Key	Value
default	doctrine.dbal.default_connection

Entity Managers

Key	Value
default	doctrine.orm.default_entity_manager

effectuée

Alors vous me direz qu'ici on n'a persisté qu'une seule entité, c'est vrai. Mais on peut tout à fait faire plusieurs persists sur différentes entités avant d'exécuter un seul flush. Le flush permet d'exécuter les requêtes les plus optimisées pour enregistrer tous nos persists.

Doctrine utilise les transactions

Pourquoi deux méthodes `$em->persist()` et `$em->flush()` ? Car cela permet entre autres de profiter des **transactions**. Imaginons que vous ayez plusieurs entités à persister en même temps. Par exemple, lorsque l'on crée un sujet sur un forum, il faut enregistrer l'entité Sujet mais aussi l'entité Message, les deux en même temps. Sans transaction, vous feriez d'abord la première requête, puis la deuxième. Logique au final. Mais imaginez que vous ayez enregistré votre Sujet, et que l'enregistrement de votre Message échoue : vous avez un sujet sans message ! Ça casse votre base de données car la relation n'est plus respectée.

Avec une transaction, les deux entités sont enregistrées *en même temps*, ce qui fait que si la deuxième échoue, alors la première est annulée, et vous gardez une base de données propre.

Concrètement, avec notre *EntityManager*, chaque `$em->persist()` est équivalent à dire : « *Garde cette entité en mémoire, tu l'enregistreras au prochain flush().* ». Et un `$em->flush()` est équivalent à : « *Ouvre une transaction et enregistre toutes les entités qui t'ont été données depuis le dernier flush().* ».

Doctrine simplifie la vie

Vous devez savoir une chose également : la méthode `$em->persist()` traite indifféremment les nouvelles entités de celles déjà en base de données. Vous pouvez donc lui passer une entité fraîchement créée comme dans notre exemple précédent, mais également une entité que vous auriez récupérée grâce à l'*EntityRepository* et que vous auriez modifiée (ou non, d'ailleurs). L'*EntityManager* s'occupe de tout, je vous disais !

Concrètement, cela veut dire que vous n'avez plus à vous soucier de faire des **INSERT INTO** dans le cas d'une création d'entité, et des **UPDATE** dans le cas d'entités déjà existantes. Exemple :

Code : PHP

```
<?php
// Depuis un contrôleur

$em = $this->getDoctrine()->getManager();

// On crée un nouvel article
$article1 = new Article;
$article1->setTitre('Mon dernier weekend');
$article1->setContenu("C'était vraiment super et on s'est bien
amusé.");
// Et on le persiste
$em->persist($article1);

// On récupère l'article d'id 5. On n'a pas encore vu cette méthode
find(), mais elle est simple à comprendre
// Pas de panique on la voit en détail dans un prochain chapitre
dédié aux Repository
$article2 = $em->getRepository('SdzBlogBundle:Article')->find(5);

// On modifie cet article, en changeant la date à la date
d'aujourd'hui
$article2->setDate(new \Datetime());

// Ici, pas besoin de faire un persist() sur $article2. En effet,
comme on a récupéré cet article via Doctrine,
// il sait déjà qu'il doit gérer cette entité. Rappelez-vous, un
persist ne sert qu'à donner la responsabilité de l'objet à
Doctrine.

// Enfin, on applique les changements à la base de données
$em->flush();
```

Le `flush()` va donc exécuter un **INSERT INTO** et un **UPDATE** tout seul. De notre côté, on a traité `$article1` exactement comme `$article2`, ce qui nous simplifie bien la vie. Comment sait-il si l'entité existe déjà ou non ? Grâce à la clé primaire de votre entité (dans notre cas, l'`id`). Si l'`id` est nul, c'est une nouvelle entité, tout simplement. 😊

Retenez bien également le fait qu'il est inutile de faire un `persist($entité)` lorsque `$entité` a été récupérée grâce à Doctrine. En effet, rappelez-vous qu'un `persist` ne fait rien d'autre que de donner la responsabilité d'un objet à Doctrine. Dans le cas de la variable `$article1` de l'exemple précédent, Doctrine ne peut pas deviner qu'il doit s'occuper de cet objet si on ne le lui dit pas ! D'où le `persist()`. Mais à l'inverse, comme c'est Doctrine qui nous a donné l'objet `$article2`, il est grand et prend tout seul la responsabilité de cet objet, inutile de le lui répéter.

Sachez également que Doctrine est assez intelligent pour savoir si une entité a été modifiée ou non. Ainsi, si dans notre exemple on ne modifiait pas `$article2`, Doctrine ne ferait pas de requête **UPDATE** inutile.

Les autres méthodes utiles de l'EntityManager

En plus des deux méthodes les plus importantes, `persist` et `flush`, l'*EntityManager* dispose de quelques méthodes intéressantes. Je ne vais vous présenter ici que les plus utilisées, mais elles sont bien sûr toutes documentées dans la [documentation officielle](#), que je vous invite fortement à aller voir.

`clear($nomEntité)` annule tous les `persist` effectués. Si le nom d'une entité est précisé (son namespace complet ou son raccourci), uniquement les `persist` sur des entités de ce type seront annulés. Voici un exemple :

Code : PHP

```
<?php
$em->persist($article);
$em->persist($commentaire);
$em->clear();
$em->flush(); // N'exécutera rien car les deux persists sont annulés
par le clear
```

detach(\$entite) annule le persist effectué sur l'entité en argument. Au prochain `flush()`, aucun changement ne sera appliquée à l'entité donc. Voici un exemple :

Code : PHP

```
<?php
$em->persist($article);
$em->persist($commentaire);
$em->detach($article);
$em->flush(); // Enregistre le $commentaire mais pas l'$article
```

contains(\$entite) retourne true si l'entité donnée en argument est gérée par l'*EntityManager* (s'il y a eu un persist sur l'entité donc). Voici un exemple :

Code : PHP

```
<?php
$em->persist($article);
var_dump($em->contains($article)); // Affiche true
var_dump($em->contains($commentaire)); // Affiche false
```

refresh(\$entite) met à jour l'entité donnée en argument dans l'état où elle est en base de données. Cela écrase et donc annule tous les changements qu'il a pu y avoir sur l'entité concernée. Voici un exemple :

Code : PHP

```
<?php
$article->setTitre('Un nouveau titre');
$em->refresh($article);
var_dump($article->getTitre()); // Affiche « Un ancien titre »
```

remove(\$entite) supprime l'entité donnée en argument de la base de données. Effectif au prochain `flush()`. Voici un exemple :

Code : PHP

```
<?php
$em->remove($article);
$em->flush(); // Exécute un DELETE sur l'$article
```

Récupérer ses entités avec un EntityRepository

Un prochaine chapitre entier est consacré aux repository, juste après dans cette partie sur Doctrine. Les repository ne sont qu'un outil pour récupérer vos entités très facilement, nous apprendrons à les maîtriser entièrement. Mais en avant première, sachez au moins récupérer une unique entité en fonction de son id.

Il faut d'abord pour cela récupérer le repository de l'entité que vous voulez. On l'a vu précédemment, voici un rappel :

Code : PHP

```
<?php
// Depuis un contrôleur

getRepository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');
```

Puis, depuis ce repository, il faut utiliser la méthode `find($id)` qui permet de retourner l'entité correspondant à l'id `$id`. Je vous invite à essayer ce code directement dans la méthode `voirAction()` de notre contrôleur Blog, là où on avait défini en dur un tableau `$article`. On pourra ainsi voir l'effet immédiatement :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...

public function voirAction($id)
{
    // On récupère le repository
```

```
$repository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');

// On récupère l'entité correspondant à l'id $id
$article = $repository->find($id);

// $article est donc une instance de
Sdz\BlogBundle\Entity\Article

// Ou null si aucun article n'a été trouvé avec l'id $id
if($article === null)
{
    throw $this->createNotFoundException('Article[id='.$id.']'
inexistant.');
}

return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
    'article' => $article
));
}
```



Allez voir le résultat sur la page `/blog/article/1` ! Vous pouvez changer l'id de l'article à récupérer dans l'URL, en fonction des articles que vous avez ajoutés plus haut depuis la méthode `ajouterAction()`.

Sachez aussi qu'il existe une autre syntaxe pour faire la même chose directement depuis l'`EntityManager`, je vous la présente afin que vous ne soyez pas surpris si vous la croisez :

Code : PHP

```
<?php

// Depuis un contrôleur

$article = $this->getDoctrine()
    ->getManager()
    ->find('SdzBlogBundle:Article', $id); // 1er
argument : le nom de l'entité
                                         // 2e argument
: l'id de l'instance à récupérer
```

Je n'en dis pas plus pour le moment, patientez jusqu'au chapitre dédié sur les repository !

En résumé

- Il faut exécuter la commande `doctrine:schema:update` pour mettre à jour la base de données et la faire correspondre à l'état actuel de vos entités ;
- Avec Symfony2, on récupère l'`EntityManager` de Doctrine2 via le service `doctrine.orm.entity_manager` ou, plus simplement depuis un contrôleur, via `$this->getDoctrine()->getManager()` ;
- L'`EntityManager` sert à **manipuler** les entités, tandis que les repositories servent à **récupérer** les entités.

Les relations entre entités avec Doctrine2

Dans ce chapitre nous allons apprendre à mettre en relation nos entités, afin de créer un ensemble cohérent.

Présentation

Présentation

Vous savez déjà stocker vos entités indépendamment les unes des autres, c'est très bien. Simplement on est rapidement limité ! L'objectif de ce chapitre est de vous apprendre à établir des relations entre les entités.

Rappelez-vous, au début de la partie sur Doctrine2 je vous avais promis des choses comme `<?php $article->getCommentaires()`. Et bien c'est cela que nous allons faire ici !

Les différents types de relations

Il y a plusieurs façon de lier des entités entre elles. En effet il n'est pas pareil de lier une multitudes de commentaires à un seul article, que de lier un membre à un seul groupe. Il existe donc plusieurs types de relations, pour répondre à plusieurs besoins concrets. Ce sont les relations OneToOne, OneToMany et ManyToMany. On les étudie juste après ces quelques notions de base à avoir.

Notions techniques d'ORM à savoir

Avant de voir en détail les relations, il faut comprendre comment elles fonctionnent. N'ayez pas peur, il y a juste deux notions à savoir avant d'attaquer.

Notion de propriétaire et d'inverse

La notion de propriétaire et d'inverse est abstraite mais importante à comprendre. *Dans une relation entre deux entités, il y a toujours une entité dite propriétaire, et une dite inverse.* Pour comprendre cette notion, il faut revenir à la vieille époque, lorsque l'on faisait nos bases de données à la main. *L'entité propriétaire est celle qui contient la référence à l'autre entité.* Attention, cette notion — à avoir en tête lors de la création des entités — n'est pas liée à votre logique métier, elle est purement technique.

Prenons un exemple simple, toujours les commentaires d'un article de blog. Vous disposez de la table « commentaire » et de la table « article ». Pour créer une relation entre ces deux tables, vous allez mettre naturellement une colonne « article_id » dans la table « commentaire ». La table « commentaire » est donc propriétaire de la relation, car c'est elle qui contient la colonne de liaison « article_id ». Assez simple au final !

 N'allez pas me créer une colonne « article_id » dans la table des commentaires ! C'est une image, de ce que vous faitiez avant. Aujourd'hui on va laisser Doctrine gérer tout ça, et on ne va jamais mettre la main dans PhpMyAdmin. Rappelez-vous : on pense objet dorénavant, et pas base de données.

Notion d'Unidirectionnalité et de Bidirectionnalité

Cette notion est également simple à comprendre : *une relation peut être à sens unique ou à double sens.* On ne va traiter dans ce chapitre que les relations à sens unique, dites unidirectionnelles. Cela signifie que vous pourrez faire `<?php $entiteProprietaire->getEntiteInverse()` (dans notre exemple `<?php $commentaire->getArticle()`), mais vous ne pourrez pas faire `<?php $entiteInverse->getEntiteProprietaire()` (pour nous, `<?php $article->getCommentaires()`). Attention, cela ne nous empêchera pas de récupérer les commentaires d'un article, on utilisera juste une autre méthode, via l'`EntityRepository`.

Cette limitation nous permet de simplifier la façon de définir les relations. Pour bien travailler avec, il suffit juste de se rappeler qu'on ne peut faire `$entiteInverse->getEntiteProprietaire()`.

Pour des cas spécifiques, ou des préférences dans votre code, cette limitation peut être contournée en utilisant les relations à double sens, dites bidirectionnelles. Je les expliquerai rapidement à la fin de ce chapitre.

Rien n'est magique

Non rien n'est magique. Je dois vous avertir qu'un `<?php $article->getCommentaires()` est vraiment sympa mais, qu'il déclenche bien sûr une requête SQL ! Lorsqu'on récupère une entité (notre `$article` par exemple), Doctrine ne récupère pas toutes les entités qui lui sont liées (les commentaires dans l'exemple), et heureusement ! Si le faisait, cela serait extrêmement lourd. Imaginez qu'on veuille juste récupérer un Article pour avoir son titre, et Doctrine nous récupère la liste des 54 commentaires, qui en plus sont liés à leurs 54 auteurs respectifs, etc. !

Doctrine utilise ce qu'on appelle le *Lazy Loading*, « chargement fainéant » en français. C'est-à-dire qu'il ne va charger les entités à l'autre bout de la relation que si vous voulez accéder à ces entités. C'est donc pile au moment où vous faites `<?php $article->getCommentaires()` que Doctrine va charger les commentaires (avec une nouvelle requête SQL donc) puis va vous les transmettre.

Heureusement pour nous, il est possible d'éviter cela ! Parce que cette syntaxe est vraiment pratique, il serait dommage de s'en priver pour cause de requêtes SQL trop nombreuses. Il faudra simplement utiliser nos propres méthodes pour charger les entités, dans lesquelles nous ferons des jointures toutes simples. L'idée est de dire à Doctrine : « Charge l'entité Article mais également tous ses Commentaires ». Avoir nos propres méthodes pour cela permet de ne les exécuter que si nous voulons vraiment avoir les Commentaires en plus de l'Article. En somme, on se garde le choix de charger ou non la relation.

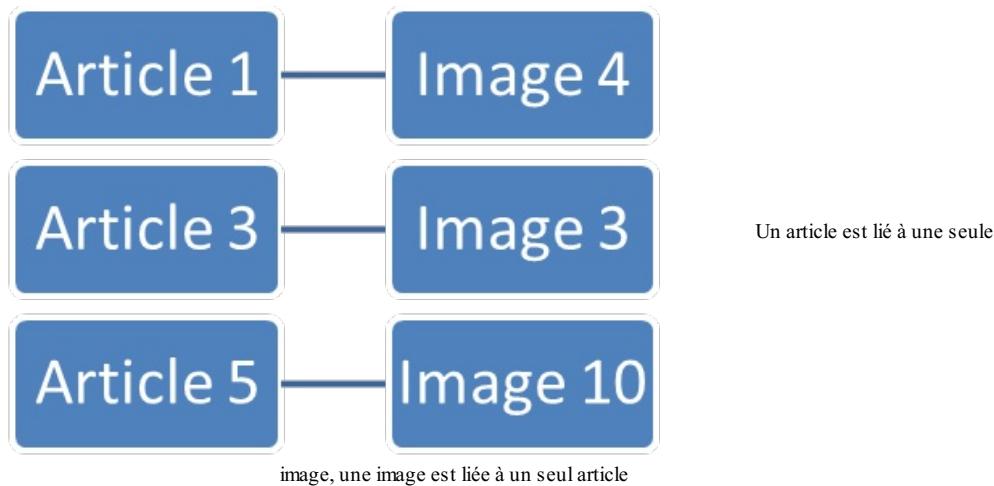
Mais nous verrons tout cela dans le prochain chapitre sur les Repository. Pour l'instant, revenons à nos relations !

Relation One-To-One

Présentation

La relation *One-To-One*, ou **1..1**, est assez classique. Elle correspond, comme son nom l'indique, à une relation unique entre deux objets.

Pour illustrer cette relation dans le cadre du blog, nous allons créer une entité `Image`. Imaginons qu'on offre la possibilité de lier une image à un article, une sorte d'icône pour illustrer un peu l'article. Si à chaque article on ne peut afficher qu'une seule image, et que chaque image ne peut être liée qu'à un seul article, alors on est bien dans le cadre d'une relation One-To-One. La figure suivante schématisé tout ça.



Tout d'abord, histoire qu'on parle bien de la même chose, créez cette entité `Image` avec au moins les attributs `url` et `alt` pour qu'on puisse l'afficher correctement. Voici la mienne :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Image.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Sdz\BlogBundle\Entity\Image
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ImageRepository")
 */
class Image
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $url
     *
     * @ORM\Column(name="url", type="string", length=255)
     */
    private $url;

    /**
     * @var string $salt
     *
     * @ORM\Column(name="alt", type="string", length=255)
     */
    private $salt;

    // Les getters et setters
}

```

Définition de la relation dans les entités

Annotation

Pour établir une relation One-To-One entre deux entités Article et Image, la syntaxe est la suivante :

Entité propriétaire, Article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image",
     * cascade={"persist"})
     */
    private $image;

    // ...
}
```

Entité inverse, Image :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Image.php

/**
 * @ORM\Entity
 */
class Image
{
    // Nul besoin de rajouter une propriété ici
    // ...
}
```

La définition de la relation est plutôt simple, mais détaillons-la bien.

Tout d'abord, j'ai choisi de définir l'entité Article comme entité propriétaire de la relation, car un Article « possède » une Image. On aura donc plus tendance à récupérer l'image à partir de l'article que l'inverse. Cela permet également de rendre indépendante l'entité Image : elle pourra être utilisée par d'autres entités que Article, de façon totalement invisible pour elle.

Ensuite, vous voyez que seule l'entité propriétaire a été modifiée, ici Article. C'est parce qu'on a une relation unidirectionnelle, rappellez-vous on peut donc faire \$article->getImage() mais pas \$image->getArticle(). Dans une relation unidirectionnelle l'entité inverse, ici Image, ne sait en fait même pas qu'elle est liée à une autre entité, ce n'est pas son rôle.

Enfin, concernant l'annotation en elle-même :

Code : Autre

```
@ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image", cascade={"persist"})
```

Il y a plusieurs choses à savoir sur cette annotation :

- Elle est incompatible avec l'annotation @ORM\Column qu'on a vu dans un chapitre précédent. En effet, l'annotation Column définit une valeur (un nombre, une chaîne de caractères, etc.), alors que OneToOne définit une relation vers une autre entité. Lorsque vous utilisez l'un, vous ne pouvez pas utiliser l'autre sur le même attribut ;
- Elle possède au moins l'attribut targetEntity, qui vaut simplement le namespace complet vers l'entité liée ;
- Elle possède d'autres attributs facultatifs, dont l'attribut cascade dont on parle un peu plus loin.



N'oubliez pas de mettre à jour la base de données avec la commande doctrine:schema:update !

Rendre une relation facultative

Par défaut, une relation est facultative, c'est-à-dire que vous pouvez avoir un Article qui n'a pas d'Image liée. C'est le comportement que nous voulons pour l'exemple : on se donne le droit d'ajouter un article sans forcément trouver une image d'illustration. Si vous souhaitez forcer la relation, il faut rajouter l'annotation `JoinColumn` et définir son attribut `nullable` à `false`, comme ceci :

Code : Autre

```
/**
 * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image")
 * @ORM\JoinColumn(nullable=false)
 */
private $image;
```

Les opérations de cascade

Parmi maintenant de l'attribut `cascade` que l'on a vu un peu plus haut. Cet attribut permet de « cascader » les opérations que l'on ferait sur l'entité Article à l'entité Image liée par la relation.

Pour prendre l'exemple le plus simple, imaginez que vous supprimez une entité Article via un `<?php $em->remove($article)`. Si vous ne précisez rien, Doctrine va supprimer l'Article mais garder l'entité Image liée. Or ce n'est pas forcément ce que vous voulez ! Si vos images ne sont liées qu'à des articles, alors la suppression de l'article doit entraîner la suppression de l'image, sinon vous avez des Images orphelines dans votre base de données. C'est le but du cascade. Attention, si vos images sont liées à des articles mais aussi à d'autres entités, alors vous ne voulez pas forcément supprimer directement l'image d'un article, car elle pourrait être liée à une autre entité.

On peut cascader des opérations de suppression, mais également de persistance. En effet, on a vu qu'il fallait persister une entité avant d'exécuter le `flush()`, afin de dire à Doctrine qu'il doit enregistrer l'entité en base de données. Cependant dans le cas d'entités liées, si on fait un `$em->persist($article)`, qu'est-ce que Doctrine doit faire pour l'entité Image contenue dans l'entité Article ? Il ne le sait pas et c'est pourquoi il faut le lui dire : soit en faisant manuellement un persist sur l'article et l'image, soit en définissant dans l'annotation de la relation qu'un persist sur Article doit se « propager » sur l'Image liée.

C'est ce que nous avons fait dans l'annotation : on a défini le cascade sur l'opération `persist`, mais pas sur l'opération `remove` (car on se réserve la possibilité d'utiliser les images pour autre chose que des articles).

Getter et setter

D'abord n'oubliez pas de définir un getter et un setter dans l'entité propriétaire, ici Article. Vous pouvez utiliser la commande `php app/console doctrine:generate:entities SdzBlogBundle:Article`, ou alors prendre ce code :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image")
     */
    private $image;

    // Vos autres attributs...

    /**
     * @param Sdz\BlogBundle\Entity\Image $image
     */
    public function setImage(\Sdz\BlogBundle\Entity\Image $image =
null)
    {
        $this->image = $image;
    }

    /**
     * @return Sdz\BlogBundle\Entity\Image
     */
    public function getImage()
    {
        return $this->image;
    }

    // Vos autres getter/setter...
}
```

Vous voyez qu'on a forcé le type de l'argument pour le setter `setImage()` : cela permet de déclencher une erreur si vous

essayez de passer un autre objet que `Image` à la méthode. Très utile pour éviter de chercher des heures l'origine d'un problème parce que vous avez passé un mauvais argument. Notez également le « = `null` » qui permet d'accepter les valeurs `null` : rappelez-vous la relation est facultative !

Prenez bien conscience d'une chose également : le getter `getImage()` retourne une instance de la classe `Image` directement. Lorsque vous avez un `Article`, disons `$article`, et que vous voulez récupérer l'URL de l'`Image` associée, il faut donc faire :

Code : PHP

```
<?php
$image = $article->getImage();
$url = $image->getUrl();

// Ou bien sûr en plus simple :
$url = $article->getImage()->getUrl();
```

Pour les curieux qui auront été voir ce qui a été fait en base de données : une colonne `image_id` a bien été ajouté à la table `article`. Cependant, ne confondez surtout pas cette colonne `image_id` avec notre attribut `image`, et gardez bien ces deux points en tête :

- 1/ L'entité `Article` ne contient pas d'attribut `image_id`.
- 2/ L'attribut `image` ne contient pas l'id de l'`Image` liée, il contient une instance de la classe `Sdz\BlogBundle\Entity\Image` qui, elle, contient un attribut `id`.



N'allez donc jamais m'écrire `$article->getImageId()`, pour récupérer l'id de l'image liée, il faut d'abord récupérer l'`Image` elle-même puis son id, comme ceci : `$article->getImage()->getId()`. Et ne faites surtout pas la confusion : une Entité n'est pas une Table.

Exemple d'utilisation

Pour utiliser cette relation, c'est très simple. Voici un exemple pour ajouter un nouvel `Article` et son `Image` depuis un contrôleur. Modifions l'action `ajouterAction()`, qui était déjà bien complète :

Code : PHP

```
<?php
// src/Sdz/Bundle/Controller/BlogController.php

// N'oubliez pas de rajouter ce use !
use Sdz\BlogBundle\Entity\Image;

// ...

public function ajouterAction()
{
    // Création de l'entité Article
    $article = new Article();
    $article->setTitre('Mon dernier weekend');
    $article->setContenu("C'était vraiment super et on s'est bien
amusé.");
    $article->setAuteur('winzou');

    // Création de l'entité Image
    $image = new Image();
    $image-
    >setUrl('http://uploads.siteduzero.com/icones/478001_479000/478657.png');
    $image->setAlt('Logo Symfony2');

    // On lie l'image à l'article
    $article->setImage($image);

    // On récupère l'EntityManager
    $em = $this->getDoctrine()->getManager();

    // Étape 1 : on persiste les entités
    $em->persist($article);

    // Étape 1 bis : si on n'avait pas défini le cascade={"persist"}, on devrait persister à la main l'entité $image
    // $em->persist($image);

    // Étape 2 : on déclenche l'enregistrement
    $em->flush();

    // ... reste de la méthode
}
```

Pour information, voici comment on pourrait modifier la vue de visualisation d'un article, pour y intégrer l'image :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #
 
{% extends "SdzBlogBundle::layout.html.twig" %}

{% block title %}
Lecture d'un article - {{ parent() }}
{% endblock %}

{% block sdzblog_body %}



## {# On vérifie qu'une image soit bien associée à l'article #} {% if article.image is not null %} {{ article.image.alt }} Par {{ article.auteur }}, le {{ article.date|date('d/m/Y') }} {{ article.contenu }} Retour à la liste Modifier l'article Supprimer l'article


```

Et voici un autre exemple, qui modifierait l'Image d'un Article déjà existant. Ici je vais prendre une méthode arbitraire mais vous savez tout ce qu'il faut pour l'implémenter réellement :

Code : PHP

```
<?php
// Dans un contrôleur, celui que vous voulez

public function modifierImageAction($id_article)
{
    $em = $this->getDoctrine()->getManager();

    // On récupère l'article
    $article = $em->getRepository('SdzBlogBundle:Article')-
>find($id_article);

    // On modifie l'URL de l'image par exemple
    $article->getImage()->setUrl('test.png');

    // On n'a pas besoin de persister notre article (si vous le
    faites aucune erreur n'est déclenchée, Doctrine l'ignore)
    // Rappelez-vous il l'est automatiquement car on l'a récupéré
    depuis Doctrine

    // Pas non plus besoin de persister l'image ici, car elle est
    également récupérée par Doctrine

    // On déclenche la modification
    $em->flush();

    return new Response('OK');
}
```

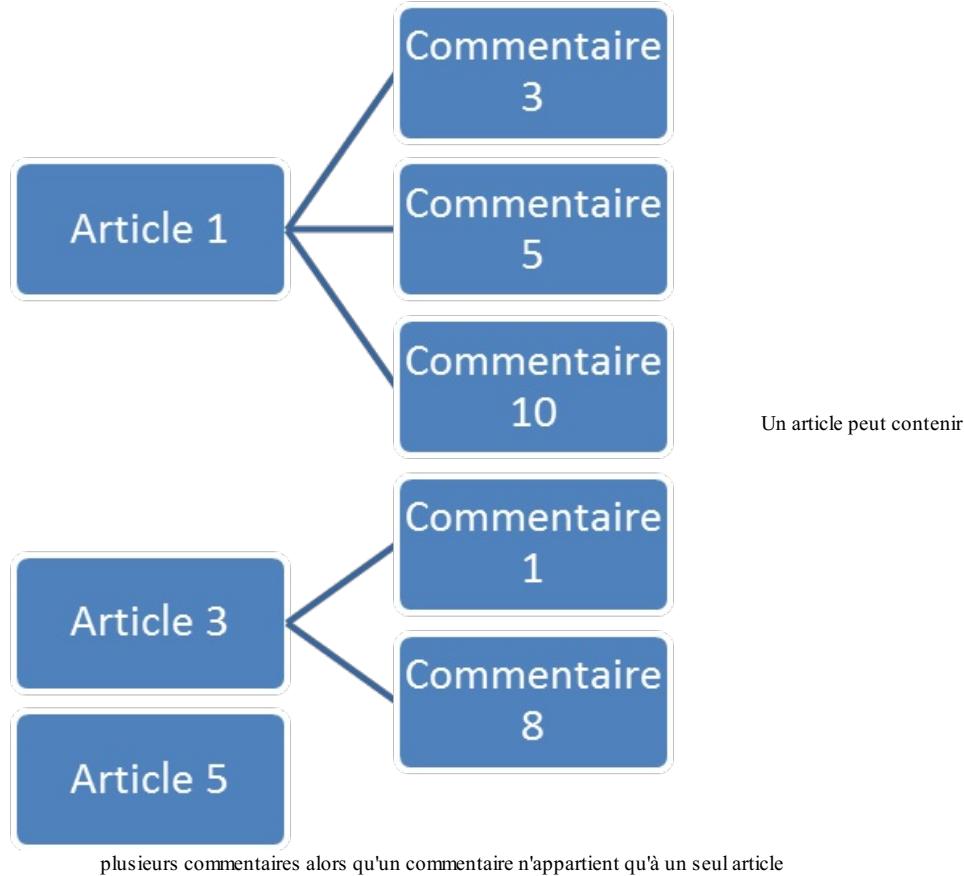
Le code parle de lui-même : gérer une relation est vraiment aisé avec Doctrine !

Relation Many-To-One

Présentation

La relation *Many-To-One*, ou **n..1**, est assez classique également. Elle correspond, comme son nom l'indique, à une relation qui permet à une entité A d'avoir une relation avec plusieurs entités B.

Pour illustrer cette relation dans le cadre de notre blog, nous allons créer une entité *Commentaire*. L'idée est de pouvoir ajouter *plusieurs commentaires à un article*, et que *chaque commentaire ne soit lié qu'à un seul article*. Nous avons ainsi plusieurs commentaires (*Many*) à lier (*To*) à un seul article (*One*). La figure suivante schématisé tout ça.



Comme précédemment, pour être sûr qu'on parle bien de la même chose, créez cette entité *Commentaire* avec au moins les attributs *auteur*, *contenu* et *date*. Voici la mienne :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Sdz\BlogBundle\Entity\Commentaire
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CommentaireRepository")
 */
class Commentaire
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $auteur
     *
     * @ORM\Column(name="auteur", type="string", length=255)
     */
    private $auteur;
}
  
```

```
    /**
 * @var text $contenu
 */
* @ORM\Column(name="contenu", type="text")
*/
private $contenu;

    /**
 * @var datetime $date
 */
* @ORM\Column(name="date", type="datetime")
*/
private $date;

public function __construct()
{
    $this->date = new \Datetime();
}

/**
 * Get id
 */
* @return integer
*/
public function getId()
{
    return $this->id;
}

/**
 * Set auteur
 */
* @param string $auteur
*/
public function setAuteur($auteur)
{
    $this->auteur = $auteur;
}

/**
 * Get auteur
 */
* @return string
*/
public function getAuteur()
{
    return $this->auteur;
}

/**
 * Set contenu
 */
* @param text $contenu
*/
public function setContenu($contenu)
{
    $this->contenu = $contenu;
}

/**
 * Get contenu
 */
* @return text
*/
public function getContenu()
{
    return $this->contenu;
}

/**
 * Set date
 */
* @param datetime $date
*/
public function setDate(\Datetime $date)
{
    $this->date = $date;
}

/**
 * Get date
 */
* @return datetime
*/
public function getDate()
{
    return $this->date;
}
```

Définition de la relation dans les entités

Annotation

Pour établir cette relation dans votre entité, la syntaxe est la suivante :

Entité propriétaire, Commentaire :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

/**
 * @ORM\Entity
 */
class Commentaire
{
    /**
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     * @ORM\JoinColumn(nullable=false)
     */
    private $article;

    // ...
}
```

Entité inverse, Article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    // Nul besoin de rajouter de propriété, ici
    // ...
}
```

L'annotation à utiliser est tout simplement `ManyToOne`.

Première remarque : l'entité propriétaire pour cette relation est `Commentaire`, et non `Article`. Pourquoi ? Parce que rappelez-vous, le propriétaire est celui qui contient la colonne référence. Ici, on aura bien une colonne `article_id` dans la table `commentaire`. En fait, de façon systématique, c'est le côté `Many` d'une relation `Many-To-One` qui est le propriétaire, vous n'avez pas le choix. Ici, on a plusieurs commentaires pour un seul article, le `Many` correspond aux commentaires, donc l'entité `Commentaire` est la propriétaire.

Deuxième remarque : j'ai volontairement rajouté l'annotation `JoinColumn` avec son attribut `nullable` à `false`, pour interdire la création d'un commentaire sans article. En effet, dans notre cas, un commentaire qui n'est rattaché à aucun article n'a pas de sens. Après attention, il se peut très bien que dans votre application vous deviez laisser la possibilité au côté `Many` de la relation d'exister sans forcément être attaché à un côté `One`.



N'oubliez pas de mettre à jour la base de données avec la commande `doctrine:schema:update` !

Getter et setter

Ajoutons maintenant le getter et le setter correspondant dans l'entité propriétaire. Comme tout à l'heure, vous pouvez utiliser la méthode `php app/console doctrine:generate:entities SdzBlogBundle:Commentaire`, ou alors mettez ceux-là :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

/**
 * @ORM\Entity
 */
class Commentaire
{
    // ...
}
```

```

class Commentaire
{
    /**
 * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
 * @ORM\JoinColumn(nullable=false)
 */
    private $article;

    // ... reste des attributs

    /**
 * Set article
 *
 * @param Sdz\BlogBundle\Entity\Article $article
 */
    public function setArticle(\Sdz\BlogBundle\Entity\Article
$article)
    {
        $this->article = $article;
    }

    /**
 * Get article
 *
 * @return Sdz\BlogBundle\Entity\Article
 */
    public function getArticle()
    {
        return $this->article;
    }

    // ... reste des getter et setter
}

```



Vous pouvez remarquer que, comme notre relation est obligatoire, il n'y a pas le « = null » dans le `setArticle()`.

Exemple d'utilisation

La méthode pour gérer une relation *ManyToOne* n'est pas très différente de celle pour une relation *OneToOne*, voyez par vous-même dans ces exemples.

Tout d'abord pour ajouter un nouvel Article et ses Commentaires, modifions encore la méthode `ajouterAction()` de notre contrôleur :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// N'oubliez pas ce use !
use Sdz\BlogBundle\Entity\Commentaire;

public function ajouterAction()
{
    // Création de l'entité Article
    $article = new Article();
    $article->setTitre('Mon dernier weekend');
    $article->setContenu("C'était vraiment super et on s'est bien
amusé.");
    $article->setAuteur('winzou');

    // Création d'un premier commentaire
    $commentaire1 = new Commentaire();
    $commentaire1->setAuteur('winzou');
    $commentaire1->setContenu('On veut les photos !');

    // Création d'un deuxième commentaire, par exemple
    $commentaire2 = new Commentaire();
    $commentaire2->setAuteur('Choupy');
    $commentaire2->setContenu('Les photos arrivent !');

    // On lie les commentaires à l'article
    $commentaire1->setArticle($article);
    $commentaire2->setArticle($article);

    // On récupère l'EntityManager
    $em = $this->getDoctrine()->getManager();

    // Étape 1 : On persiste les entités
    $em->persist($article);
    // Pour cette relation pas de cascade, car elle est définit
}

```

```

dans l'entité Commentaire et non Article
// On doit donc tout persister à la main ici
$em->persist($commentaire1);
$em->persist($commentaire2);

// Étape 2 : On déclenche l'enregistrement
$em->flush();

// ... reste de la méthode
}

```

Pour information, voici comment on pourrait modifier l'action `voirAction()` du contrôleur pour passer non seulement l'article à la vue, mais également ses commentaires (lignes 20, 21 et 26) :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function voirAction($id)
{
    // On récupère l'EntityManager
    $em = $this->getDoctrine()
        ->getEntityManager();

    // On récupère l'entité correspondant à l'id $id
    $article = $em->getRepository('SdzBlogBundle:Article')
        ->find($id);

    if($article === null)
    {
        throw $this->createNotFoundException('Article[id='.$id.]'
inexistant.');
    }

    // On récupère la liste des commentaires
    $liste_commentaires = $em-
>getRepository('SdzBlogBundle:Commentaire')
        ->findAll();

    // Puis modifiez la ligne du render comme ceci, pour prendre en
    // compte l'article :
    return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
        'article'      => $article,
        'liste_commentaires' => $liste_commentaires
    ));
}

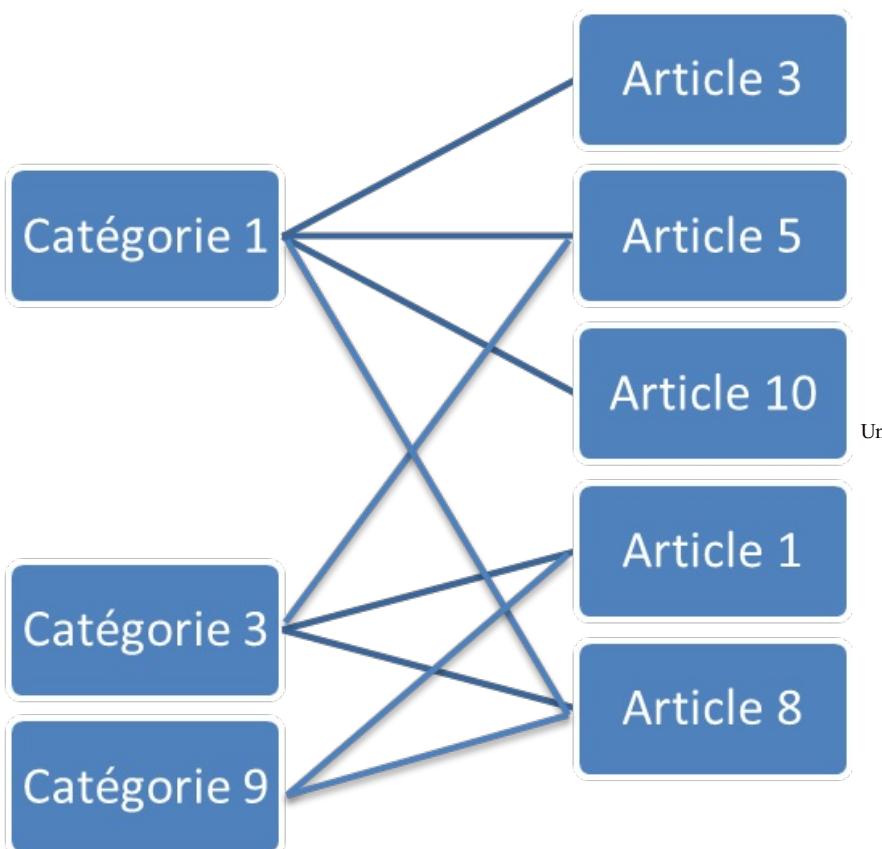
```

Ici vous pouvez voir qu'on a utilisé la méthode `findAll()`, qui récupère tous les commentaires et pas seulement ceux de l'article actuel. Il faudra bien sûr modifier ce comportement, nous le ferons dans le prochain chapitre, avec le repository qui permet de personnaliser nos requêtes. Et bien entendu, il faudrait adapter la vue si vous voulez afficher la liste des commentaires que nous venons de lui passer.

Relation Many-To-Many Présentation

La relation *Many-To-Many*, ou **n..n**, correspond à une relation qui permet à plein d'objets d'être en relation avec plein d'autres !

Prenons l'exemple cette fois-ci des articles de notre blog, répartis dans des catégories. *Un article peut appartenir à plusieurs catégories. À l'inverse, une catégorie peut contenir plusieurs articles.* On a donc une relation *Many-To-Many* entre Article et Catégorie. La figure suivante schématisé tout ça.



article peut appartenir à plusieurs catégories et plusieurs catégories peuvent appartenir à plusieurs articles

Cette relation est particulière dans le sens où Doctrine va devoir créer une table intermédiaire. En effet, avec la méthode traditionnelle en base de données, comment feriez-vous pour faire ce genre de relation ? Vous avez une table *article*, une autre table *categorie*, mais vous avez surtout besoin d'une table *article_categorie* qui fait la liaison entre les deux ! Cette table de liaison ne contient que deux colonnes : *article_id* et *categorie_id*. Cette table intermédiaire, vous ne la connaîtrez pas : elle n'apparaît pas dans nos entités, et c'est Doctrine qui la crée et qui la gère tout seul !

 Je vous ai parlé de cette table intermédiaire pour que vous compreniez comment Doctrine fonctionne. Cependant attention, nous sommes d'accord que vous devez totalement oublier cette notion de table intermédiaire lorsque vous manipulez des objets (les entités) ! J'insiste sur le fait que si vous voulez utiliser Doctrine, alors il faut le laisser gérer la base de données tout seul : vous utilisez des objets, lui utilise une base de données, chacun son travail.

Encore une fois, pour être sûr que l'on parle bien de la même chose, créez cette entité *Categorie* avec au moins un attribut nom. Voici la mienne :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Categorie.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Sdz\BlogBundle\Entity\Categorie
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CategorieRepository")
 */
class Categorie
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $nom
     *
     * @ORM\Column(name="nom", type="string", length=255)
     */
  
```

```

/*
    private $nom;

    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set nom
     *
     * @param string $nom
     * @return Categorie
     */
    public function setNom($nom)
    {
        $this->nom = $nom;
        return $this;
    }

    /**
     * Get nom
     *
     * @return string
     */
    public function getNom()
    {
        return $this->nom;
    }
}

```

Définition de la relation dans les entités

Annotation

Pour établir cette relation dans vos entités, la syntaxe est la suivante.

Entité propriétaire, Article :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Categorie",
     * cascade={"persist"})
     */
    private $categories;

    // ...
}

```

Entité inverse, Categorie :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Categorie.php

/**
 * @ORM\Entity
 */
class Categorie
{
    // Nul besoin d'ajouter une propriété, ici
    // ...
}

```

J'ai mis Article comme propriétaire de la relation. C'est un choix que vous pouvez faire comme bon vous semble, ici. Mais bon, récupérer les catégories d'un article se fera assez souvent, alors que récupérer les articles d'une catégorie moins. Et puis, pour récupérer les articles d'une catégorie, on aura sûrement besoin de personnaliser la requête, donc on le fera de toute façon depuis le *CategorieRepository*.

Getter et setter

Dans ce type de relation, il faut soigner un peu plus l'entité propriétaire. Tout d'abord, on a pour la première fois un attribut (ici \$categories) qui contient une liste d'objets. C'est parce qu'il contient une liste d'objets qu'on a mis le nom de cet attribut au pluriel. Les listes d'objets avec Doctrine2 ne sont pas de simples tableaux, mais des ArrayCollection, il faudra donc définir l'attribut comme tel dans le constructeur. Un ArrayCollection est un objet utilisé par Doctrine2, qui a toutes les propriétés d'un tableau normal. Vous pouvez faire un foreach dessus, et le traiter comme n'importe quel tableau. Il dispose juste de quelques méthodes supplémentaires très pratiques, que nous verrons.

Ensuite, le getter est classique et s'appelle `getCategories()`. Par contre, c'est les setters qui vont différer un peu. En effet, \$categories est une liste de catégories, mais au quotidien ce qu'on va faire c'est ajouter une à une des catégories à cette liste. Il nous faut donc une méthode `addCategorie()` (sans « s », on n'ajoute qu'une seule catégorie à la fois) et non une `setCategories()`. Du coup, il nous faut également une méthode pour supprimer une catégorie de la liste, que l'on appelle `removeCategorie()`.

Ajoutons maintenant le getter et les setter correspondants dans l'entité propriétaire, Article. Comme tout à l'heure, vous pouvez utiliser la méthode `php app/console doctrine:generate:entities SdzBlogBundle:Article`, ou alors reprendre ce code :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Categorie",
     * cascade={"persist"})
     */
    private $categories;

    // ... vos autres attributs

    // Comme la propriété $categories doit être un ArrayCollection,
    // souvenez-vous
    // On doit la définir dans un constructeur :
    public function __construct()
    {
        // Si vous aviez déjà un constructeur, rajoutez juste cette
        // ligne :
        $this->categories = new
        \Doctrine\Common\Collections\ArrayCollection();
    }

    /**
     * Add categories
     *
     * @param Sdz\BlogBundle\Entity\Categorie $categories
     */
    public function addCategorie(\Sdz\BlogBundle\Entity\Categorie
$categorie) // addCategorie sans « s » !
    {
        // Ici, on utilise l'ArrayCollection vraiment comme un tableau,
        // avec la syntaxe []
        $this->categories[] = $categorie;
    }

    /**
     * Remove categories
     *
     * @param Sdz\BlogBundle\Entity\Categorie $categories
     */
    public function removeCategorie(\Sdz\BlogBundle\Entity\Categorie
$categorie) // removeCategorie sans « s » !
    {
        // Ici on utilise une méthode de l'ArrayCollection, pour
        // supprimer la catégorie en argument
        $this->categories->removeElement($categorie);
    }

    /**
     * Get categories
     *
     * @return Doctrine\Common\Collections\Collection
     */
}
```

```
public function getCategories() // Notez le « s », on récupère
{
    return $this->categories;
}

// ... vos autres getter/setter
}
```



N'oubliez pas de mettre à jour la base de données avec la commande `doctrine:schema:update`.

Remplissons la base de données avec les Fixtures

Avant de voir un exemple, j'aimerais vous faire ajouter quelques catégories en base de données histoire d'avoir de quoi jouer avec. Pour cela, petit aparté, nous allons faire une Fixture Doctrine ! Cela va nous permettre d'utiliser le bundle qu'on a installé lors du chapitre sur Composer.

Les fixtures Doctrine permettent de remplir la base de données avec un jeu de données que nous allons définir. Cela permet de pouvoir tester avec des vraies données, sans devoir les retaper à chaque fois : on les inscrit une fois pour toute, et ensuite elles sont toutes insérées en base de données en une seule commande.

Tout d'abord, créons notre fichier de fixture pour l'entité `Categorie`. Les fixtures d'un bundle se trouvent dans le répertoire `DataFixtures/ORM` (ou ODM pour des documents). Voici à quoi ressemble notre fixture `Categories` :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/DataFixtures/ORM/Categories.php

namespace Sdz\BlogBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Sdz\BlogBundle\Entity\Categorie;

class Categories implements FixtureInterface
{
    // Dans l'argument de la méthode load, l'objet $manager est
    // l'EntityManager
    public function load(ObjectManager $manager)
    {
        // Liste des noms de catégorie à ajouter
        $noms = array('Symfony2', 'Doctrine2', 'Tutorial', 'Évènement');

        foreach($noms as $i => $nom)
        {
            // On crée la catégorie
            $liste_categories[$i] = new Categorie();
            $liste_categories[$i]->setNom($nom);

            // On la persiste
            $manager->persist($liste_categories[$i]);
        }

        // On déclenche l'enregistrement
        $manager->flush();
    }
}
```

C'est tout ! On peut dès à présent insérer ces données dans la base de données. Voici donc la commande à exécuter :

Code : Console

```
C:\wamp\www\Symfony>php app/console doctrine:fixtures:load
Careful, database will be purged. Do you want to continue Y/N ?y
> purging database
> loading Sdz\BlogBundle\DataFixtures\ORM\Categories
```

Et voilà ! Les quatre catégories définies dans le fichier de fixture sont maintenant enregistrées en base de données, on va pouvoir s'en servir dans nos exemples. Par la suite, on rajoutera d'autres fichiers de fixture pour insérer d'autres entités en base de données : la commande les traitera tous l'un après l'autre.



Attention, comme vous avez pu le voir, l'exécution de la commande Doctrine pour insérer les fixtures vide totalement la base de données avant d'insérer les nouvelles données. Si vous voulez ajouter les fixtures en plus des données déjà



présentes, il faut ajouter l'option `--append` à la commande précédente. Cependant, c'est rarement ce que vous voulez, car à la prochaine exécution des fixtures, vous allez insérer une nouvelle fois les mêmes catégories...

Exemples d'utilisation

Voici un exemple pour ajouter un article existant à plusieurs catégories existantes. Je vous propose de mettre ce code dans notre méthode `modifierAction()` par exemple :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

class BlogController extends Controller
{
    // ...

    // Ajout d'un article existant à plusieurs catégories existantes :
    public function modifierAction($id)
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
            ->getManager();

        // On récupère l'entité correspondant à l'id $id
        $article = $em->getRepository('SdzBlogBundle:Article')
            ->find($id);

        if($article === null)
        {
            throw $this->createNotFoundException('Article[id='.$id.'] inexistant.');
        }

        // On récupère toutes les catégories :
        $liste_categories = $em-
            >getRepository('SdzBlogBundle:Categorie')
            ->findAll();

        // On boucle sur les catégories pour les lier à l'article
        foreach($liste_categories as $categorie)
        {
            $article->addCategorie($categorie);
        }

        // Inutile de persister l'article, on l'a récupéré avec
        // Doctrine
        // Etape 2 : On déclenche l'enregistrement
        $em->flush();

        return new Response('OK');
    }
}
```

Je vous mets un exemple concret d'application pour que vous puissiez vous représenter l'utilisation de la relation dans un vrai cas d'utilisation. Mais sinon les seules lignes qui concernent vraiment l'utilisation de notre relation ManyToMany sont les lignes 29 à 32 : la boucle sur les catégories pour ajouter chaque catégorie une à une à l'article en question.

Voici un autre exemple pour enlever toutes les catégories d'un article. Modifions la méthode `supprimerAction()` pour l'occasion :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

class BlogController extends Controller
{
    // ...

    // Suppression des catégories d'un article :
    public function supprimerAction($id)
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
            ->getManager();
```

```

// On récupère l'entité correspondant à l'id $id
$article = $em->getRepository('SdzBlogBundle:Article')
    ->find($id);

if($article === null)
{
    throw $this->createNotFoundException('Article[id='.$id.']'
inexistant.');
}

// On récupère toutes les catégories :
$liste_categories = $em-
>getRepository('SdzBlogBundle:Categorie')
    ->findAll();

// On enlève toutes ces catégories de l'article
foreach($liste_categories as $categorie)
{
    // On fait appel à la méthode removeCategorie() dont on a
    parlé plus haut
    // Attention ici $categorie est bien une instance de
    Categorie, et pas seulement un id
    $article->removeCategorie($categorie);
}

// On n'a pas modifié les catégories : inutile de les persister

// On a modifié la relation Article - Categorie
// Il faudrait persister l'entité propriétaire pour persister la
relation
// Or l'article a été récupéré depuis Doctrine, inutile de le
persister

// On déclenche la modification
$em->flush();

return new Response('OK');
}
}

```

Encore une fois, je vous ai mis un code complet, mais ce qui nous intéresse dans le cadre de la relation ce n'est que les lignes 30 à 35.

Enfin, voici un dernier exemple pour afficher les catégories d'un article dans la vue. Regardez plus particulièrement les lignes 21 à 27 :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

{% extends "SdzBlogBundle::layout.html.twig" %}

{% block title %}
    Lecture d'un article - {{ parent() }}
{% endblock %}

{% block sdzblog_body %}

    <h2>
        {# On vérifie qu'une image soit bien associée à l'article #}
        {% if article.image is not null %}
            
        {% endif %}

        {{ article.titre }}
    </h2>
    <i>Par {{ article.auteur }}, le {{ article.date|date('d/m/Y') }}</i>

    {% if article.categories.count > 0 %}
        - Catégories :
        {% for categorie in article.categories %}
            {{ categorie.nom }}
            {% if not loop.last %} - {% endif %}
        {% endfor %}
        {% endif %}

        <div class="well">
            {{ article.contenu }}
        </div>

        <p>
            <a href="{{ path('sdzblog_accueil') }}" class="btn">
                <i class="icon-chevron-left"></i>
            </a>
        </p>
    {% endif %}

```

```

        Retour à la liste
    </a>
    <a href="{{ path('sdzblog_modifier', {'id': article.id}) }}"
    class="btn">
        <i class="icon-edit"></i>
        Modifier l'article
    </a>
    <a href="{{ path('sdzblog_supprimer', {'id': article.id}) }}"
    class="btn">
        <i class="icon-trash"></i>
        Supprimer l'article
    </a>
</p>

{%
    endblock
%}

```

Vous voyez qu'on accède aux catégories d'un article grâce à l'attribut `categories` de l'article, tout simplement. En Twig, cela signifie `{{ article.categories }}`, en PHP on ferait `$article->getCategories()`. Il suffit ensuite de parcourir ce tableau grâce à une boucle, et d'en faire ce que l'on veut.

Relation Many-To-Many avec attributs

Présentation

La relation *Many-To-Many* qu'on vient de voir peut suffire dans bien des cas, mais elle est en fait souvent incomplète pour les besoins d'une application.

Pour illustrer ce manque, rien de tel qu'un exemple : considérons l'entité `Produit` d'un site e-commerce ainsi que l'entité `Commande`. Une commande contient plusieurs produits, et bien entendu un même produit peut être dans différentes commandes. On a donc bien une relation *Many-To-Many*. Voyez-vous le manque ? Lorsqu'un utilisateur ajoute un produit à une commande, où met-on la quantité de ce produit qu'il veut ? Si je veux 3 exemplaires de Harry Potter, où mettre cette quantité ? Dans l'entité `Commande` ? Non cela n'a pas de sens. Dans l'entité `Produit` ? Non, cela n'a pas de sens non plus. Cette quantité est un attribut de la relation qui existe entre `Produit` et `Commande`, et non un attribut de `Produit` ou de `Commande`.

Il n'y a pas de moyen simple de gérer les attributs d'une relation avec Doctrine. Pour cela, il faut esquiver en créant simplement une entité intermédiaire qui va représenter la relation, appellons-la `CommandeProduit`. Et c'est dans cette entité que l'on mettra les attributs de relation, comme notre quantité. Ensuite il faut bien entendu mettre en relation cette entité intermédiaire avec les deux autres entités d'origine, `Commande` et `Produit`. Pour cela, il faut logiquement faire : `Commande One-To-Many CommandeProduit Many-To-One Produit`. En effet, une commande (One) peut avoir plusieurs relations avec des produits (Many), plusieurs `CommandeProduit` donc ! La relation est symétrique pour les produits.

Attention, dans le titre de cette section j'ai parlé de la relation *Many-To-Many avec attributs*, mais il s'agit bien en fait de deux relations *Many-To-One* des plus normales, soyons d'accord. On ne va donc rien apprendre dans ce prochain paragraphe, car on sait déjà faire une *Many-To-One*, mais c'est une astuce qu'il faut bien connaître et savoir utiliser, donc prenons le temps de bien la comprendre.

J'ai pris l'exemple de produits et de commandes car c'est plus intuitif pour comprendre l'enjeu et l'utilité de cette relation. Cependant, pour rester dans le cadre de notre blog, on va faire une relation entre des `Article` et des `Compétence`, et l'attribut de la relation sera le niveau. L'idée est de pouvoir afficher sur chaque article la liste des compétences utilisées (Symfony2, Doctrine2, Formulaire, etc.) avec le niveau dans chaque compétence (Débutant, Avisé et Expert). On a alors l'analogie suivante :

- Article <=> Commande
- ArticleCompetence <=> Commande_Produit
- Compétence <=> Produit

Et donc : `Article One-To-Many ArticleCompetence Many-To-One Competence`.

Pour cela, créez d'abord cette entité `Compétence`, avec au moins un attribut `nom`. Voici la mienne :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Competence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Sdz\BlogBundle\Entity\Competence
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CompetenceRepository")
 */
class Competence
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     */

```

```

 * @ORM\GeneratedValue(strategy="AUTO")
 */
private $id;

/**
 * @var string $nom
 *
 * @ORM\Column(name="nom", type="string", length=255)
 */
private $nom;

/**
 * Get id
 *
 * @return integer
 */
public function getId()
{
    return $this->id;
}

/**
 * Set nom
 *
 * @param string $nom
 * @return Competence
 */
public function setNom($nom)
{
    $this->nom = $nom;
}

/**
 * Get nom
 *
 * @return string
 */
public function getNom()
{
    return $this->nom;
}
}

```

Définition de la relation dans les entités

Annotation

Tout d'abord, on va créer notre entité de relation (notre ArticleCompetence) comme ceci :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/ArticleCompetence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class ArticleCompetence
{
    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     */
    private $article;

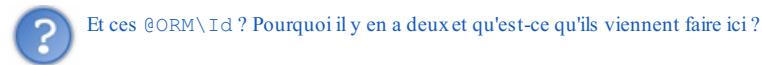
    /**
     * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Competence")
     */
    private $competence;

    /**
     * @ORM\Column()
     */
    private $niveau; // Ici j'ai un attribut de relation « niveau »

    // ... vous pouvez ajouter d'autres attributs bien entendu
}

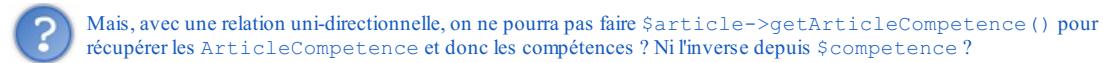
```

Comme les côtés *Many* des deux relations *Many-To-One* sont dans `ArticleCompetence`, cette entité est l'entité propriétaire des deux relations.



Très bonne question. Comme toute entité, notre `ArticleCompetence` se doit d'avoir un identifiant. C'est obligatoire pour que Doctrine puisse la gérer par la suite. Depuis le début nous avons rajouté un attribut `id` qui était en auto-incrémentation, on ne s'en occupait pas trop donc. Ici c'est différent, comme une `ArticleCompetence` correspond à un unique couple `Article/Competence` (pour chaque couple `Article/Competence`, il n'y a qu'une seule `ArticleCompetence`), on peut se servir de ces deux attributs pour former l'identifiant de cette entité.

Pour cela, il suffit de définir `@ORM\Id` sur les deux colonnes, et Doctrine saura mettre une clé primaire sur ces deux colonnes puis les gérer comme n'importe quel autre identifiant. Encore une fois, merci Doctrine !



En effet, et c'est pourquoi la prochaine section de ce chapitre traite des relations bidirectionnelles ! En attendant, pour notre relation *One-To-Many-To-One*, continuons simplement sur une relation unidirectionnelle.

Sachez quand même que vous pouvez éviter une relation bi-directionnelle ici en utilisant simplement la méthode `findByCommande($commande->getId())` (pour récupérer les produits d'une commande) ou `findByProduit($produit->getId())` (pour l'inverse) du repository `CommandeProduitRepository`.

L'intérêt de la bi-directionnelle ici est lorsque vous voulez afficher une liste des commandes avec leurs produits. Dans la boucle sur les commandes, vous n'allez pas faire appel à une méthode du repository qui va générer une requête par boucle, il faudra passer par un `$commande->getCommandeProduits()`. Nous le verrons plus loin dans ce chapitre.



Getter et setter

Comme d'habitude les getters et setters doivent se définir dans l'entité propriétaire. Ici, rappelez-vous nous sommes en présence de deux relations *Many-To-One* dont la propriétaire est l'entité `ArticleCompetence`. Nous avons donc deux getters et deux setters classiques à écrire. Vous pouvez les générer avec la commande `doctrine:generate:entities SdzBlogBundle:ArticleCompetence`, ou mettre le code suivant :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleCompetence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class ArticleCompetence
{
    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     */
    private $article;

    /**
     * @ORM\Column()
     */
    private $niveau; // Ici j'ai un attribut de relation « niveau »

    // ... les autres attributs

    // Getter et setter pour l'entité Article
    public function setArticle(\Sdz\BlogBundle\Entity\Article
        $article)
    {
        $this->article = $article;
    }
}
```

```

    }
    public function getArticle()
    {
        return $this->article;
    }

    // Getter et setter pour l'entité Competence
    public function setCompetence(\Sdz\BlogBundle\Entity\Competence
$competence)
    {
        $this->competence = $competence;
    }
    public function getCompetence()
    {
        return $this->competence;
    }

    // On définit le getter/setter de l'attribut « niveau »
    public function setNiveau($niveau)
    {
        $this->niveau = $niveau;
    }
    public function getNiveau()
    {
        return $this->niveau;
    }

    // ... les autres getter/setter si vous en avez
}

```

Remplissons la base de données

Comme précédemment, on va d'abord rajouter des compétences en base de données grâce aux fixtures. Pour faire une nouvelle fixture, il suffit de créer un nouveau fichier dans le répertoire DataFixtures/ORM dans le bundle. Je vous invite à créer le fichier `src/Sdz/Bundle/DataFixtures/ORM/Competences.php`:

Code : PHP

```

<?php
// src/Sdz/Bundle/DataFixtures/ORM/Competences.php

namespace Sdz\BlogBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Sdz\BlogBundle\Entity\Competence;

class Competences implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        // Liste des noms de compétences à ajouter
        $noms = array('Doctrine', 'Formulaire', 'Twig');

        foreach($noms as $i => $nom)
        {
            // On crée la compétence
            $liste_competences[$i] = new Competence();
            $liste_competences[$i]->setNom($nom);

            // On la persiste
            $manager->persist($liste_competences[$i]);
        }

        // On déclenche l'enregistrement
        $manager->flush();
    }
}

```

Et maintenant on peut exécuter la commande :

Code : Console

```

C:\wamp\www\Symfony>php app/console doctrine:fixtures:load
Careful, database will be purged. Do you want to continue Y/N ?y
> purging database
> loading Sdz\BlogBundle\DataFixtures\ORM\Categories
> loading Sdz\BlogBundle\DataFixtures\ORM\Competences

```

Vous pouvez voir qu'après avoir tout vidé, Doctrine a inséré les fixtures Categories puis nos fixtures Compétences. Tout est prêt !

Exemple d'utilisation

La manipulation des entités dans une telle relation est un peu plus compliquée, surtout sans la bidirectionnalité. Mais on peut tout de même s'en sortir. Tout d'abord, voici un exemple pour créer un nouvel article contenant plusieurs compétences ; mettons ce code dans la méthode ajouterAction() :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// N'oubliez pas ce use évidemment
use Sdz\BlogBundle\Entity\ArticleCompetence;

class BlogController extends Controller
{
    // ...

    public function ajouterAction()
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
                    ->getManager();

        // Création de l'entité Article
        $article = new Article();
        $article->setTitre('Mon dernier weekend');
        $article->setContenu("C'était vraiment super et on s'est bien
        amusé.");
        $article->setAuteur('winzou');

        // Dans ce cas, on doit créer effectivement l'article en bdd
        // pour lui assigner un id
        // On doit faire ça pour pouvoir enregistrer les
        ArticleCompetence par la suite
        $em->persist($article);
        $em->flush(); // Maintenant, $article a un id définit

        // Les compétences existent déjà, on les récupère depuis la bdd
        $liste_competences = $em-
            >getRepository('SdzBlogBundle:Competence')
                ->findAll(); // Pour l'exemple, notre
        Article contient toutes les Competences

        // Pour chaque compétence
        foreach($liste_competences as $i => $competence)
        {
            // On crée une nouvelle "relation entre 1 article et 1
            compétence"
            $articleCompetence[$i] = new ArticleCompetence;

            // On la lie à l'article, qui est ici toujours le même
            $articleCompetence[$i]->setArticle($article);
            // On la lie à la compétence, qui change ici dans la boucle
            foreach
                $articleCompetence[$i]->setCompetence($competence);

            // Arbitrairement, on dit que chaque compétence est requis au
            niveau 'Expert'
            $articleCompetence[$i]->setNiveau('Expert');

            // Et bien sûr, on persiste cette entité de relation,
            propriétaire des deux autres relations
            $em->persist($articleCompetence[$i]);
        }

        // On déclenche l'enregistrement
        $em->flush();

        // ... reste de la méthode
    }
}
```

Et voici un autre exemple pour récupérer les compétence et leur niveau à partir d'un article donné. Je vous propose de modifier la méthode voirAction() :

Code : PHP

```
<?php
```

```
// src/Sdz/BlogBundle/Controller/BlogController.php

class BlogController extends Controller
{
    // ...

    public function voirAction($id)
    {
        // On récupère l'EntityManager
        $em = $this->getDoctrine()
            ->getManager();

        // On récupère l'entité correspondant à l'id $id
        $article = $em->getRepository('SdzBlogBundle:Article')
            ->find($id);

        if($article === null)
        {
            throw $this->createNotFoundException('Article[id='.$id.']'
inexistant.');
        }

        // On récupère les articleCompetence pour l'article $article
        $liste_articleCompetence = $em-
>getRepository('SdzBlogBundle:ArticleCompetence')
            ->findByArticle($article->getId());

        // Puis modifiez la ligne du render comme ceci, pour prendre en
        // compte les articleCompetence :
        return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
            'article'      => $article,
            'liste_articleCompetence' => $liste_articleCompetence,
            // ... et évidemment les autres variables que vous pouvez avoir
        ));
    }
}
```

Et le code de la vue correspondante :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

{% extends "SdzBlogBundle::layout.html.twig" %}

{% block title %}
    Lecture d'un article - {{ parent() }}
{% endblock %}

{% block sdzblog_body %}

    <h2>
        {% On vérifie qu'une image soit bien associée à l'article %}
        {% if article.image is not null %}
            
        {% endif %}

        {{ article.titre }}
    </h2>
    <i>Par {{ article.auteur }}, le {{ article.date|date('d/m/Y') }}</i>

    {% if article.categories.count > 0 %}
        - Catégories :
        {% for categorie in article.categories %}
            {{ categorie.nom }}
            {% if not loop.last %} - {% endif %}
        {% endfor %}
        {% endif %}

    <div class="well">
        {{ article.contenu }}
    </div>

    {% if liste_articleCompetence|length > 0 %}
    <div>
        Compétences utilisées dans cet article :
        <ul>
        {% for articleCompetence in liste_articleCompetence %}
            <li>{{ articleCompetence.competence.nom }} : {{ articleCompetence.niveau }}</li>
        {% endfor %}
        </ul>
    </div>
    {% endif %}

```

```

<p>
    <a href="{{ path('sdzblog_accueil') }}" class="btn">
        <i class="icon-chevron-left"></i>
        Retour à la liste
    </a>
    <a href="{{ path('sdzblog_modifier', { 'id': article.id}) }}" class="btn">
        <i class="icon-edit"></i>
        Modifier l'article
    </a>
    <a href="{{ path('sdzblog_supprimer', { 'id': article.id}) }}" class="btn">
        <i class="icon-trash"></i>
        Supprimer l'article
    </a>
</p>

{%
    endblock %
}

```

C'est un exemple simple bien sûr 😊.

Attention, dans cet exemple la méthode `findByArticle()` utilisée dans le contrôleur ne sélectionne que les `ArticleCompetence`. Donc, lorsque dans la boucle dans la vue, on fait `{ { articleCompetence.competence } }`, en réalité Doctrine va effectuer une requête pour récupérer la `Competence` associée à cette `ArticleCompetence`. C'est bien sûr une horreur, car il va faire une requête... par itération dans le `for` ! Si vous avez 20 compétences attachées à l'article, cela ferait 20 requêtes, inimaginable.

Pour charger les `Competence` en même temps que les `ArticleCompetence` dans le contrôleur, et ainsi ne plus faire de requête dans la boucle, il faut faire une méthode à nous dans le repository de `ArticleCompetence`. On voit tout ça dans le chapitre suivant dédié aux repository. N'utilisez donc jamais cette technique, attendez le prochain chapitre ! La seule différence dans le contrôleur sera d'utiliser une autre méthode que `findByArticle()`, et la vue ne changera même pas.

Les relations bidirectionnelles

Présentation

Vous avez vu que jusqu'ici, nous n'avons jamais modifié l'entité inverse d'une relation, mais seulement l'entité propriétaire. Toutes les relations que l'on vient de faire sont donc des relations unidirectionnelles.

Leur avantage est de définir la relation d'une façon très simple. Mais l'inconvénient est de ne pas pouvoir récupérer l'entité propriétaire depuis l'entité inverse, le fameux `<?php $entiteInverse->getEntiteProprietaire()` (pour nous, `<?php $article->getCommentaires()` par exemple). Je dis inconvénient, mais vous avez pu constater que cela ne nous a pas du tout empêché de faire ce qu'on voulait ! À chaque fois, on a réussi à ajouter, lister, modifier nos entités et leurs relations.

Mais dans certains cas, avoir une relation bidirectionnelle est bien utile. Nous allons les voir rapidement dans cette section. Sachez que la documentation l'explique également très bien : vous pourrez vous renseigner sur [le chapitre sur la création des relations](#), puis [celui sur leur utilisation](#).

Définition de la relation dans les entités

Pour étudier la définition d'une relation bidirectionnelle, nous allons étudier une relation *ManyToOne*. Souvenez-vous bien de cette relation, dans sa version unidirectionnelle, pour pouvoir attaquer sa version bidirectionnelle dans les meilleures conditions.

Nous allons ici construire une relation bidirectionnelle de type *Many-To-One*, basée sur notre exemple `Article-Commentaire`. Mais la méthode est exactement la même pour les relations de type *One-To-One* ou *ManyToMany*.



Je pars du principe que vous avez déjà une relation unidirectionnelle fonctionnelle. Si ce n'est pas votre cas, mettez-la en place avant de lire la suite du paragraphe, car nous n'allons voir que les rajouts à faire.

Annotation

Alors, attaquons la gestion d'une relation bidirectionnelle. L'objectif de cette relation est de rendre possible l'accès à l'entité propriétaire depuis l'entité inverse. Avec une unidirectionnelle cela n'est pas possible car on ne rajoute pas d'attribut dans l'entité inverse, ce qui signifie que l'entité inverse ne sait même pas qu'elle fait partie d'une relation.

La première étape consiste donc à rajouter un attribut, et son annotation, à notre entité inverse `Article`:

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Commentaire",

```

```

mappedBy="article"
*/
private $commentaires; // Ici commentaires prend un "s", car un
// article a plusieurs commentaires !
// ...
}

```

Bien entendu, je vous dois des explications sur ce que l'on vient de faire.

Commençons par l'annotation. L'inverse d'un *ManyToOne* est... un *OneToMany*, tout simplement ! Il faut donc utiliser l'annotation *OneToMany* dans l'entité inverse. Je rappelle que le propriétaire d'une relation *ManyToOne* est toujours le côté *Many*, donc lorsque vous voyez l'annotation *ManyToOne*, vous êtes forcément du côté propriétaire. Ici on a un *OneToMany*, on est bien du côté inverse.

Ensuite les paramètres de cette annotation. Le *targetEntity* est évident, il s'agit toujours de l'entité à l'autre bout de la relation, ici notre entité *Commentaire*. Le *mappedBy* correspond lui à l'attribut de l'entité propriétaire (*Commentaire*) qui pointe vers l'entité inverse (*Article*) : c'est le *private \$article* de l'entité *Commentaire*. Il faut le renseigner pour que l'entité inverse soit au courant des caractéristiques de la relation : celles-ci sont définies dans l'annotation de l'entité propriétaire.

Il faut également adapter l'entité propriétaire, pour lui dire que maintenant la relation est de type bidirectionnelle et non plus unidirectionnelle. Pour cela il faut rajouter le paramètre *inversedBy* dans l'annotation *ManyToOne* :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

/**
 * @ORM\Entity
 */
class Commentaire
{
    /**
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article",
     * inversedBy="commentaires")
     * @ORM\JoinColumn(nullable=false)
     */
    private $article;

    // ...
}

```

Ici, nous avons seulement rajouté le paramètre *inversedBy*. Il correspond au symétrique du *mappedBy*, c'est-à-dire à l'attribut de l'entité inverse (*Article*) qui pointe vers l'entité propriétaire (*Commentaire*). C'est donc l'attribut *commentaires*.

Tout est bon côté annotation, maintenant il faut également rajouter les getters et setters dans l'entité inverse bien entendu.

Getter et setter

On part d'une relation unidirectionnelle fonctionnelle, donc les getter et setter de l'entité propriétaire sont bien définis.

Dans un premier temps, rajoutons assez logiquement le getter et le setter dans l'entité inverse. On vient de lui rajouter un attribut, il est normal que le getter et le setter aillent de paire. Comme nous sommes du côté *One* d'un *OneToMany*, l'attribut *commentaires* est un *ArrayCollection*. C'est donc un *addCommentaire* / *removeCommentaire* / *getCommentaires* qu'il nous faut. Encore une fois, vous pouvez le générer avec `doctrine:generate:entities SdzBlogBundle:Article`, ou alors vous pouvez mettre ce code :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 * @ORM\Entity
 */
class Article
{
    /**
     * @ORM\OneToMany(targetEntity="Sdz\BlogBundle\Entity\Commentaire",
     * mappedBy="article")
     */
    private $commentaires; // Ici commentaires prend un « s », car un
// article a plusieurs commentaires !

    // ... vos autres attributs

    public function __construct()

```

```

{
    // Rappelez-vous, on a un attribut qui doit contenir un
    // ArrayCollection, on doit l'initialiser dans le constructeur
    $this->commentaires = new
    \Doctrine\Common\Collections\ArrayCollection();
}

public function addCommentaire(\Sdz\BlogBundle\Entity\Commentaire
$commentaire)
{
    $this->commentaires[] = $commentaire;
    return $this;
}

public function
removeCommentaire(\Sdz\BlogBundle\Entity\Commentaire $commentaire)
{
    $this->commentaires->removeElement($commentaire);
}

public function getCommentaires()
{
    return $this->commentaires;
}

// ...
}

```

Maintenant, il faut nous rendre compte d'un petit manque. Voici une petite problématique, lisez bien ce code :

Code : PHP

```

<?php
// Création des entités
$article = new Article;
$commentaire = new Commentaire;

// On lie le commentaire à l'article
$article->addCommentaire($commentaire);

```



Que retourne `$commentaire->getArticle()` ?

Rien ! En effet pour qu'un `$commentaire->getArticle()` retourne un article, il faut d'abord le lui définir en appelant `$commentaire->setArticle($article)`, c'est logique !



Si vous ne voyez pas pourquoi Doctrine n'a pas rempli l'attribut `article` de l'objet `$commentaire`, il faut revenir aux fondamentaux. Vous êtes en train d'écrire du PHP, `$article` et `$commentaire` sont des objets PHP, tels qu'ils existaient bien avant la naissance de Doctrine. Pour que l'attribut `article` soit défini, il faut absolument faire appel au setter `setArticle()` car c'est le seul qui accède à cet attribut (qui est en `private`). Dans le petit exemple que je vous ai mis, on n'a pas exécuté de fonction Doctrine : à aucun moment il n'a pu intervenir et éventuellement exécuter le `setArticle()` 😊

C'est logique en soi, mais du coup dans notre code cela va être moins beau : il faut en effet lier le commentaire à l'article *et* l'article au commentaire. Comme ceci :

Code : PHP

```

<?php
// Création des entités
$article = new Article;
$commentaire = new Commentaire;

// On lie le commentaire à l'article
$article->addCommentaire($commentaire);

// On lie l'article au commentaire
$commentaire->setArticle($article);

```

Mais ces deux méthodes étant intimement liées, on doit en fait les imbriquer. En effet, laisser le code dans l'état est possible, mais imaginez qu'un jour vous oubliez d'appeler l'une des deux méthodes ; votre code ne sera plus cohérent. Et un code non cohérent est un code qui a des risques de contenir des bugs. La bonne méthode est donc simplement de faire appel à l'une des méthodes depuis l'autre. Voici concrètement comme le faire en modifiant les setters dans l'une des deux entités :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Article

/**
 * @ORM\Entity
 */
class Article
{
    // ...

    public function addCommentaire(\Sdz\BlogBundle\Entity\Commentaire
$commentaire)
    {
        $this->commentaires[] = $commentaire;
        $commentaire->setArticle($this); // On ajoute ceci
        return $this;
    }

    public function
removeCommentaire(\Sdz\BlogBundle\Entity\Commentaire $commentaire)
    {
        $this->commentaires->removeElement($commentaire);
        // Et si notre relation était facultative (nullable=true, ce
qui n'est pas notre cas ici attention) :
        // $commentaire->setArticle(null);
    }

    // ...
}

```

Notez qu'ici j'ai modifié un côté de la relation (l'inverse en l'occurrence), mais surtout pas les deux ! En effet, si `addCommentaire()` exécute `setArticle()`, qui exécute à son tour `addCommentaire()`, qui... etc. On se retrouve avec une boucle infinie.

Bref, l'important est de se prendre un côté (propriétaire ou inverse n'a pas d'importance), et de l'utiliser. Par utiliser j'entends que dans le reste du code (contrôleur, service, etc.), il faudra ici exécuter `$article->addCommentaire()` qui garde la cohérence entre les deux entités. Il ne faudra pas exécuter `$commentaire->setArticle()`, car lui ne garde pas la cohérence ! Retenez : *on modifie le setter d'un côté, et on utilise ensuite ce setter là*. C'est simple, mais important à respecter.

Pour conclure

Le chapitre sur les relations Doctrine touche ici à sa fin.

Pour maîtriser les relations que nous venons d'apprendre, il faut vous entraîner à les créer et à les manipuler. N'hésitez donc pas à créer des entités d'entraînement, et à voir leur comportement dans les relations.

Si vous voulez plus d'information sur les fixtures que l'on a rapidement abordé lors de ce chapitre, je vous invite à lire la page de la documentation du bundle : [http://symfony.com/fr/doc/current/bund \[...\] le/index.html](http://symfony.com/fr/doc/current/bund [...] le/index.html)

Rendez-vous au prochain chapitre pour apprendre à récupérer les entités depuis la base de données à votre guise, grâce aux repository !

En résumé

- Les relations Doctrine révèlent toute la puissance de l'ORM ;
- Dans une relation entre deux entités, l'une est propriétaire de la relation et l'autre est inverse. Cette notion est purement technique ;
- Une relation est dite unidirectionnelle si l'entité inverse n'a pas d'attribut la liant à l'entité propriétaire. On met en place une relation bidirectionnelle lorsqu'on a besoin de cet attribut dans l'entité inverse (ce qui arrivera pour certains formulaires, etc.).

Récupérer ses entités avec Doctrine2

L'une des principales fonctions de la couche Modèle dans une application MVC, c'est la récupération des données. Récupérer des données n'est pas toujours évident, surtout lorsqu'on veut récupérer seulement certaines données, les classer selon des critères, etc. Tout ceci se fait grâce aux *repository*, que nous étudions dans ce chapitre. Bonne lecture !

Le rôle des Repository

On s'est déjà rapidement servi de quelques repository, donc vous devriez sentir leur utilité, mais il est temps de théoriser un peu.

Définition

Un *repository* centralise tout ce qui touche à la récupération de vos entités. Concrètement donc, vous ne devez pas faire la moindre requête SQL ailleurs que dans un repository, c'est la règle. On va donc y construire des méthodes pour récupérer une entité par son id, pour récupérer une liste d'entités suivant un critère spécifique, etc. Bref, à chaque fois que vous devez récupérer des entités dans votre base de données, vous utiliserez le repository de l'entité correspondante.

Rappelez-vous, il existe un repository par entité. Cela permet de bien organiser son code. Bien sûr, cela n'empêche pas qu'un repository utilise plusieurs types d'entité, dans le cas d'une jointure par exemple.

Les repository ne fonctionnent pas par magie, ils utilisent en réalité directement l'*EntityManager* pour faire leur travail. Vous le verrez, parfois nous ferons directement appel à l'*EntityManager* depuis des méthodes du repository.

Les méthodes de récupération des entités

Depuis un repository, il existe deux moyens de récupérer les entités : en utilisant du DQL et en utilisant le *QueryBuilder*.

Le Doctrine Query Language (DQL)

Le DQL n'est rien d'autre que du SQL adapté à la vision par objets que Doctrine utilise. Il s'agit donc de faire ce qu'on a l'habitude de faire, des requêtes textuelles comme celle-ci par exemple :

Code : SQL

```
SELECT a FROM SdzBlogBundle:Article a
```

Vous venez de voir votre première requête DQL. Retenez le principe : avec une requête qui n'est rien d'autre que du texte, on effectue le traitement voulu.

Le QueryBuilder

Le *QueryBuilder* est un moyen plus nouveau. Comme son nom l'indique, il sert à construire une requête, par étape. Si l'intérêt n'est pas évident au début, son utilisation se révèle vraiment pratique ! Voici la même requête que précédemment, mais en utilisant le *QueryBuilder* :

Code : PHP

```
<?php
$QueryBuilder
->select('a')
->from('SdzBlogBundle:Article', 'a');
```

Un des avantages est qu'il est possible de construire la requête en plusieurs fois. Ainsi, vous pouvez développer une méthode qui rajoute une condition à une requête, par exemple pour sélectionner tous les membres actifs (qui se sont connectés depuis moins d'un mois par exemple). Comme cette condition risque de servir souvent, dans plusieurs requêtes, avant vous deviez la réécrire à chaque fois. Avec le *QueryBuilder*, vous pourrez faire appel à la même méthode, sans réécrire la condition. Pas de panique on verra des exemples dans la suite du chapitre !

Les méthodes de récupération de base

Définition

Vos repository héritent de la classe `Doctrine\ORM\EntityRepository`, qui propose déjà quelques méthodes très utiles pour récupérer des entités. Ce sont ces méthodes là que nous allons voir ici.

Les méthodes normales

Il existe quatre méthodes, que voici (tous les exemples sont depuis un contrôleur).

La méthode `find($id)` récupère tout simplement l'entité correspondant à l'id `$id`. Dans le cas de notre `ArticleRepository`, elle retourne une instance d'`Article`. Exemple :

Code : PHP

```
<?php
$repository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');

$article_5 = $repository->find(5);
// $article 5 est une instance de Sdz\BlogBundle\Entity\Article
```

La méthode `findAll()` retourne toutes les entités. Le format du retour est un simple Array, que vous pouvez parcourir (avec un `foreach` par exemple) pour utiliser les objets qu'il contient. Exemple :

Code : PHP

```
<?php
$repository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');

$liste_articles = $repository->findAll();

foreach($liste_articles as $article)
{
    // $article est une instance de Article
    echo $article->getContenu();
}
```

Ou dans une vue Twig, si l'on a passé la variable `$liste_articles` au template :

Code : HTML

```
<ul>
    {% for article in liste_articles %}
        <li>{{ article.contenu }}</li>
    {% endfor %}
</ul>
```

La méthode `findBy()` est un peu plus intéressante. Comme `findAll()` elle permet de retourner une liste d'entité, sauf qu'elle est capable d'effectuer un filtre pour ne retourner que les entités correspondant à un critère. Elle peut aussi trier les entités, et même n'en récupérer qu'un certain nombre (pour une pagination). Exemple :

Code : PHP

```
<?php
//Syntaxe :
findBy(array $critères, array $orderBy = null, $limite = null,
$offset = null);

getRepository = $this->getDoctrine()
    ->getManager()
    ->getRepository('SdzBlogBundle:Article');

$liste_articles = $repository->findBy(array('auteur' => 'winzou'),
array('date' => 'desc'),
5,
0);

foreach($liste_articles as $article)
{
    // $article est une instance de Article
    echo $article->getContenu();
}
```

Cet exemple va récupérer toutes les entités ayant comme auteur « winzou » en les classant par date décroissante et en en sélectionnant cinq (5) à partir du début (0). Elle retourne un Array également. Vous pouvez mettre plusieurs entrées dans le tableau des critères, afin d'appliquer plusieurs filtres.

La méthode `findOneBy(array $critères)` fonctionne sur le même principe que la méthode `findBy()`, sauf qu'elle ne retourne qu'une seule entité. Les arguments `orderBy`, `limite` et `offset` n'existent donc pas. Exemple :

Code : PHP

```
<?php
$repository = $this->getDoctrine()
    ->getManager()
```

```

->getRepository('SdzBlogBundle:Article');

$article = $repository->findOneBy(array('titre' => 'Mon dernier
weekend'));
// $article est une instance de Article

```

Ces méthodes permettent de couvrir pas mal de besoin. Mais pour aller plus loin encore, Doctrine nous offre deux autres méthodes magiques.

Les méthodes magiques

Vous connaissez le principe des [méthodes magiques](#), comme `__call()` qui émule des méthodes. Ces méthodes émulées n'existent pas dans la classe, elle sont prises en charge par `__call()` qui va exécuter du code en fonction du nom de la méthode appelée.

Voici les deux méthodes gérées par `__call()` dans les repository.

`findByX($valeur)`

En remplaçant « X » par le nom d'une propriété de votre entité. Dans notre cas, pour l'entité `Article`, nous avons donc plusieurs méthodes : `findByTitre()`, `findByDate()`, `findByAuteur()`, `findByContenu()`, etc.

Cette méthode fonctionne comme `findBy()`, sauf que vous ne pouvez mettre qu'un seul critère, celui du nom de la méthode.

Attention la limite de cette méthode est que vous ne pouvez pas utiliser les arguments pour trier, ni pour mettre une limite.

Code : PHP

```

<?php
getRepository = $this->getDoctrine()
->getManager()
->getRepository('SdzBlogBundle:Article');

$liste_articles = $repository->findByAuteur('winzou');
// $liste_articles est un Array qui contient tous les articles
écrits par winzou

```

`findOneByX($valeur)`

En remplaçant « X » par le nom d'une propriété de votre entité. Dans notre cas, pour l'entité `Article`, nous avons donc plusieurs méthodes : `findOneByTitre()`, `findOneByDate()`, `findOneByAuteur()`, `findOneByContenu()`, etc.

Cette méthode fonctionne comme `findOneBy()`, sauf que vous ne pouvez mettre qu'un seul critère, celui du nom de la méthode.

Code : PHP

```

<?php
getRepository = $this->getDoctrine()
->getManager()
->getRepository('SdzBlogBundle:Article');

$article = $repository->findOneByTitre('Mon dernier weekend');
// $article est une instance d'Article

```

Toutes ces méthodes permettent de récupérer vos entités dans la plupart des cas. Simplement, elles montrent rapidement leurs limites lorsqu'on doit faire des jointures, ou effectuer des conditions plus complexes. Pour cela — et cela nous arrivera très souvent — il faudra faire nos propres méthodes de récupération.

Les méthodes de récupération personnelles

La théorie

Pour effectuer nos propres méthodes, il faut bien comprendre comment fonctionne Doctrine2 pour effectuer ses requêtes. Il faut notamment distinguer trois types d'objets qui vont nous servir, et qu'il ne faut pas confondre : le `QueryBuilder`, la `Query` et les résultats.

Le QueryBuilder

On l'a déjà vu rapidement, le `QueryBuilder` permet de construire une `Query`, mais il n'est pas une `Query` !

Pour récupérer un `QueryBuilder`, on peut utiliser simplement l'`EntityManager`. En effet, il dispose d'une méthode `createQueryBuilder()` qui nous retournera une instance de `QueryBuilder`. L'`EntityManager` est accessible depuis un `repository` en utilisant l'attribut `_em` d'un `repository`, soit `<?php $this->_em`. Le code complet pour récupérer un

QueryBuilder neuf depuis une méthode d'un repository est donc <?php \$this->_em->createQueryBuilder().

Cependant cette méthode nous retourne un QueryBuilder vide, c'est-à-dire sans rien de prédéfini. C'est dommage, car lorsqu'on récupère un QueryBuilder depuis un repository, c'est que l'on veut faire une requête sur l'entité gérée par ce repository. Donc si l'on pouvait définir la partie `SELECT article FROM SdzBlogBundle:Article` sans trop d'effort, ça serait bien pratique car ce qui est intéressant c'est le reste de la requête. Heureusement, le repository contient également une méthode `createQueryBuilder($alias)` qui utilise la méthode de l'EntityManager, mais en définissant pour nous le SELECT et le FROM. Vous pouvez jeter un oeil à cette méthode `createQueryBuilder()` pour comprendre.

L'alias en argument de la méthode est le raccourci que l'on donne à l'entité du repository. On utilise souvent la première lettre du nom de l'entité, dans notre exemple de l'Article cela serait donc un « a ».

Beaucoup de théorie, passons donc à la pratique ! Pour bien comprendre la différence QueryBuilder / Query, ainsi que la récupération du QueryBuilder, rien de mieux qu'un exemple. Nous allons recréer la méthode `findAll()` dans notre repository Article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\EntityRepository;

/**
 * ArticleRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class ArticleRepository extends EntityRepository
{
    public function myfindAll()
    {
        $queryBuilder = $this->createQueryBuilder('a');

        // Méthode équivalente, mais plus longue :
        $queryBuilder = $this->_em->createQueryBuilder()
            ->select('a')
            ->from($this->_entityName, 'a'); // Dans un repository, $this->_entityName est le namespace de l'entité gérée
        // Ici, il vaut donc Sdz\BlogBundle\Entity\Article

        // On a fini de construire notre requête

        // On récupère la Query à partir du QueryBuilder
        $query = $queryBuilder->getQuery();

        // On récupère les résultats à partir de la Query
        $resultats = $query->getResult();

        // On retourne ces résultats
        return $resultats;
    }
}
```

Cette méthode `myfindAll()` retourne exactement le même résultat qu'un `findAll()`, c'est-à-dire un tableau de toutes les entités Article dans notre base de données. Vous pouvez le voir, faire une simple requête est très facile. Pour mieux le visualiser, je vous propose la même méthode sans les commentaires et en raccourcie :

Code : PHP

```
<?php
public function myfindAll()
{
    return $this->createQueryBuilder('a')
        ->getQuery()
        ->getResult();
}
```

Simplissime non ? 😊

Et bien sûr pour récupérer les résultats depuis un contrôleur, le code est le suivant :

Code : PHP

```
<?php
public function testAction()
{
```

```

    $liste_articles = $this->getDoctrine()
        ->getEntityManager()
        ->getRepository('SdzBlogBundle:Article')
        ->myfindAll();

    // Reste de la méthode du contrôleur
}

```

Sauf que pour l'instant on a juste récupéré le QueryBuilder, on n'a pas encore joué avec lui. Il dispose de plusieurs méthodes afin de construire notre requête. Il y a une ou plusieurs méthode(s) par partie de requête : le WHERE, le ORDER BY, le FROM, etc. Elles n'ont rien de compliqué, voyez-le dans les exemples suivants.

Commençons par une méthode équivalente au `find($id)` de base, pour nous permettre de manipuler le `where()` et le `setParameter()`.

Code : PHP

```

<?php
// Dans un repository

public function myFindOne($id)
{
    // On passe par le QueryBuilder vide de l'EntityManager pour
    // l'exemple
    $qb = $this->_em->createQueryBuilder();

    $qb->select('a')
        ->from('SdzBlogBundle:Article', 'a')
        ->where('a.id = :id')
        ->setParameter('id', $id);

    return $qb->getQuery()
        ->getResult();
}

```

Vous connaissez le principe des paramètres, qui est le même qu'avec PDO. On définit un paramètre dans la requête avec `:nom_du_paramètre`, puis on attribut une valeur à ce paramètre avec la méthode `setParameter('nom_du_paramètre', $valeur)`.

Voici un autre exemple pour utiliser le `andWhere()` ainsi que le `orderBy()`. Créons une méthode pour récupérer tous les articles écrits par un auteur avant une année donnée :

Code : PHP

```

<?php
// Depuis un repository

public function findByAuteurAndDate($auteur, $annee)
{
    // On utilise le QueryBuilder créé par le repository directement
    // pour gagner du temps
    // Plus besoin de faire le select() ni le from() par la suite
    // donc
    $qb = $this->createQueryBuilder('a');

    $qb->where('a.auteur = :auteur')
        ->setParameter('auteur', $auteur)
        ->andWhere('a.date < :annee')
        ->setParameter('annee', $annee)
        ->orderBy('a.date', 'DESC');

    return $qb->getQuery()
        ->getResult();
}

```

Maintenant voyons un des avantages du QueryBuilder. Vous vous en souvenez, je vous avais parlé d'une méthode pour centraliser une condition par exemple. Voyons donc une application de ce principe, en considérant que la condition « *articles postés durant l'année en cours* » est une condition dont on va se resservir souvent. Il faut donc en faire une méthode, que voici :

Code : PHP

```

<?php
// Depuis un repository

public function whereCurrentYear(\Doctrine\ORM\QueryBuilder $qb)
{
    $qb->andWhere('a.date BETWEEN :debut AND :fin')
        ->setParameter('debut', new \Datetime(date('Y').'-01-01')) // 

```

```

Date entre le 1er janvier de cette année
->setParameter('fin', new \Datetime(date('Y') . '-12-31)); // Et
le 31 décembre de cette année

return $qb;
}

```

Vous notez donc que cette méthode ne traite pas une Query, mais bien uniquement le QueryBuilder. C'est en ça qu'il est très pratique, car faire cette méthode sur une requête en texte simple est possible, mais compliqué. Il aurait fallu voir si le WHERE était déjà présent dans la requête, si oui mettre un AND au bon endroit, etc. Bref, pas simple.

Pour utiliser cette méthode, voici la démarche :

Code : PHP

```

<?php
// Depuis un repository

public function myFind()
{
    $qb = $this->createQueryBuilder('a');

    // On peut rajouter ce qu'on veut avant
    $qb->where('a.auteur = :auteur')
        ->setParameter('auteur', 'winzou');

    // On applique notre condition
    $qb = $this->whereCurrentYear($qb);

    // On peut rajouter ce qu'on veut après
    $qb->orderBy('a.date', 'DESC');

    return $qb->getQuery()
        ->getResult();
}

```

Voilà, vous pouvez dorénavant appliquer cette condition à n'importe laquelle de vos requêtes en construction.

Je ne vous ai pas listé toutes les méthodes du QueryBuilder, il en existe bien d'autres. Pour cela, vous devez absolument mettre la page suivante dans vos favoris : [http://www.doctrine-project.org/docs/o \[...\] -builder.html](http://www.doctrine-project.org/docs/o [...] -builder.html). Ouvrez-la et gardez-la sous la main à chaque fois que vous voulez faire une requête à l'aide du QueryBuilder, c'est la référence !

La Query

Vous l'avez vu, la Query est l'objet à partir duquel on extrait les résultats. Il n'y a pas grand chose à savoir sur cet objet en lui-même, car il ne permet pas grand chose à part récupérer les résultats. Il sert en fait surtout à la gestion du cache des requêtes. Un prochain chapitre est à venir sur ce cache de requêtes.

Mais détaillons tout de même les différentes façons d'extraire les résultats de la requêtes. Ces différentes manières sont toutes à maîtriser, car elles concernent chacune un type de requête.

getResult()

Exécute la requête et retourne un tableau contenant les résultats sous forme d'objet. Vous récupérez ainsi une liste des objets, sur lesquels vous pouvez faire des opérations, des modifications, etc.

Même si la requête ne retourne qu'un seul résultat, cette méthode retourne un tableau.

Code : PHP

```

<?php
$entites = $qb->getQuery()->getResult();

foreach($entites as $entite)
{
    // $entite est une instance d'Article pour notre exemple
    $entite->getAttribut();
}

```

getArrayResult()

Exécute la requête et retourne un tableau contenant les résultats sous forme de tableau. Comme avec getResult(), vous récupérez un tableau même s'il n'y a qu'un seul résultat. Mais dans ce tableau, vous n'avez pas vos objets d'origine, vous avez des simples tableaux. Cette méthode est utilisée lorsque vous ne voulez que lire vos résultats, sans y apporter de modification. Elle est dans ce cas plus rapide que son homologue getResult().

Code : PHP

```
<?php
$entites = $qb->getQuery()->getArrayResult();

foreach($entites as $entite)
{
    // $entite est un tableau
    // Faire $entite->getAttribut() est impossible. Vous devez faire :
    $entite['attribut'];
}
```

Heureusement Twig est intelligent : {{ entite.attribut }} exécute \$entite->getAttribut() si \$entite est un objet, et exécute \$entite['attribut'] sinon. Du point de vue de Twig, vous pouvez utiliser getResult() ou getArrayResult() indifféremment.

getScalarResult()

Exécute la requête et retourne un tableau contenant les résultats sous forme de valeur. Comme avec getResult(), vous récupérez un tableau même s'il n'y a qu'un seul résultat.

Mais dans ce tableau, un résultat est une valeur, non un tableau de valeur (getArrayResult) ou un objet de valeur (getResult). Cette méthode est donc utilisée lorsque vous ne sélectionnez qu'une seule valeur dans la requête, par exemple : `SELECT COUNT(*) FROM ...` Ici, la valeur est la valeur du COUNT.

Code : PHP

```
<?php
$entites = $qb->getQuery()->getScalarResult();

foreach($entites as $valeur)
{
    // $valeur est la valeur de ce qui a été sélectionné : un nombre,
    // un texte, etc.
    $valeur;

    // Faire $valeur->getAttribut() ou $valeur['attribut'] est
    // impossible
}
```

getOneOrNullResult()

Exécute la requête et retourne un seul résultat, ou null si pas de résultat. Cette méthode retourne donc une instance de l'entité (ou null) et non un tableau d'entités comme getResult().

Cette méthode déclenche une exception Doctrine\ORM\NonUniqueResultException si la requête retourne plus d'un seul résultat. Il faut donc l'utiliser si l'une de vos requêtes n'est pas censée retourner plus d'un résultat : déclencher une erreur plutôt que de laisser courir permet d'anticiper des futurs bugs !

Code : PHP

```
<?php
$entite = $qb->getQuery()->getOneOrNullResult();

// $entite est une instance d'Article dans notre exemple
// Ou null si la requête ne contient pas de résultat

// Et une exception a été déclenchée si plus d'un résultat
```

getSingleResult()

Exécute la requête et retourne un seul résultat. Cette méthode est exactement la même que getOneOrNullResult(), sauf qu'elle déclenche une exception Doctrine\ORM\NoResultException si aucun résultat.

C'est une méthode très utilisée, car faire des requêtes qui ne retournent qu'un unique résultat est très fréquent.

Code : PHP

```
<?php
$entite = $qb->getQuery()->getSingleResult();

// $entite est une instance d'Article dans notre exemple
```

```
// Une exception a été déclenchée si plus d'un résultat
// Une exception a été déclenchée si pas de résultat
```

getSingleScalarResult()

Exécute la requête et retourne une seule valeur, et déclenche des exceptions si pas de résultat ou plus d'un résultat.

Cette méthode est très utilisée également pour des requêtes du type **SELECT COUNT(*) FROM Article**, qui ne retourne qu'une seule ligne de résultat, et une seule valeur dans cette ligne.

Code : PHP

```
<?php
$valeur = $qb->getQuery()->getSingleScalarResult();

// $valeur est directement la valeur du COUNT dans la requête
exemple

// Une exception a été déclenchée si plus d'un résultat
// Une exception a été déclenchée si pas de résultat
```

execute()

Exécute la requête. Cette méthode est utilisée principalement pour exécuter des requêtes qui ne retournent pas de résultats (des **UPDATE**, **INSERT INTO**, etc.).

Cependant, toutes les autres méthodes que nous venons de voir ne sont en fait que des raccourcis vers cette méthode `execute()`, en changeant juste le mode d'hydratation des résultats (objet, tableau, etc.).

Code : PHP

```
<?php
// Exécute un UPDATE par exemple :
$query->execute();

// Voici deux méthodes strictement équivalentes :
$resultats = $query->getArrayResult();
// Et :
$resultats = $query->execute(array(), Query::HYDRATE_ARRAY);

// Le premier argument de execute() est un tableau de paramètres
// Vous pouvez aussi passer par la méthode setParameter(), au choix

// Le deuxième argument de execute() est ladite méthode
d'hydratation
```

Pensez donc à bien choisir votre façon de récupérer les résultats à chacune de vos requêtes.

Utilisation du Doctrine Query Language (DQL)

Le DQL est une sorte de SQL adapté à l'ORM Doctrine2. Il permet de faire des requêtes un peu à l'ancienne, en écrivant une requête en chaîne de caractères (en opposition au QueryBuilder).

Pour écrire une requête en DQL, il faut donc oublier le QueryBuilder, on utilisera seulement l'objet Query. Et la méthode pour récupérer les résultats sera la même. Le DQL n'a rien de compliqué, et il est très bien documenté.

La théorie

Pour créer une requête en utilisant du DQL, il faut utiliser la méthode `createQuery()` de l'EntityManager :

Code : PHP

```
<?php
// Depuis un repository
public function myFindAllDQL()
{
    $query = $this->em->createQuery('SELECT a FROM
SdzBlogBundle:Article a');
    $resultats = $query->getResult();

    return $resultats;
```

```
}
```

Regardons de plus près la requête DQL en elle-même :

Code : SQL

```
SELECT a FROM SdzBlogBundle:Article a
```

Tout d'abord, vous voyez que l'on n'utilise pas de table. On a dit qu'on pensait objet et non plus base de données ! Il faut donc utiliser dans les FROM et les JOIN le nom des entités. Soit en utilisant le nom raccourci comme on l'a fait, soit le namespace complet de l'entité. De plus, il faut toujours donner un alias à l'entité, ici on a mis « a ». On met souvent la première lettre de l'entité, même si ce n'est absolument pas obligatoire.

Ensuite, vous imaginez bien qu'il ne faut pas sélectionner un à un les attributs de nos entités, cela n'aurait pas de sens. Une entité Article avec le titre renseigné mais pas la date ? Ce n'est pas logique. C'est pourquoi on sélectionne simplement l'alias, ici « a », ce qui sélectionne en fait tous les attributs d'un Article. L'équivalent d'une étoile (*) en SQL donc.

 Sachez qu'il est tout de même possible de ne sélectionner qu'une partie d'un objet, en faisant « a.titre » par exemple. Mais vous ne recevez alors qu'un tableau contenant les attributs sélectionnés, et non un objet. Vous ne pouvez donc pas modifier/supprimer/etc. l'objet, puisque c'est un tableau. Cela sert dans des requêtes particulières, mais la plupart du temps on sélectionnera bien tout l'objet.

Faire des requêtes en DQL n'a donc rien de compliqué. Lorsque vous les faites, gardez bien sous la main [la page de la documentation sur le DQL](#) pour en connaître la syntaxe. En attendant, je peux vous montrer quelques exemples afin que vous ayez une idée globale du DQL.

 Pour tester rapidement vos requêtes DQL, sans avoir à les implémenter dans une méthode de votre repository, Doctrine2 nous simplifie la vie grâce à la commande `doctrine:query:dql`. Cela vous permet de faire quelques tests afin de construire ou de vérifier vos requêtes, à utiliser sans modération donc ! Je vous invite dès maintenant à exécuter la commande suivante :

```
php app/console doctrine:query:dql "SELECT a FROM SdzBlogBundle:Article a"
```

Exemples

Pour faire une jointure :

Code : SQL

```
SELECT a, u FROM Article a JOIN a.utilisateur u WHERE u.age = 25
```

Pour utiliser une fonction SQL :

Code : SQL

```
SELECT a FROM Article a WHERE TRIM(a.auteur) = 'winzou'
```

Pour sélectionner seulement un attribut (attention les résultats seront donc sous forme de tableaux et non d'objets) :

Code : SQL

```
SELECT a.titre FROM Article a WHERE a.id IN(1, 3, 5)
```

Et bien sûr vous pouvez également utiliser des paramètres :

Code : PHP

```
<?php
public function myFindDQL($id)
{
    $query = $this->_em->createQuery('SELECT a FROM Article a WHERE
a.id = :id');
    $query->setParameter('id', $id);
    return $query->getSingleResult(); // Utilisation de
getSingleResult car la requête ne doit retourner qu'un seul
résultat
```

}

Utiliser les jointures dans nos requêtes

Pourquoi utiliser les jointures ?

Je vous en ai déjà parlé dans le chapitre précédent sur les relations entre entités. Lorsque vous utilisez la syntaxe `<?php $entiteA->getEntiteB()`, Doctrine exécute une requête afin de charger les entités B qui sont liées à l'entité A.

L'objectif est donc d'avoir la maîtrise sur quand charger juste l'entité A, et quand charger l'entité A avec ses entités B liées. Nous avons déjà vu le premier cas, par exemple un `$repositoryA->find($id)` ne récupère qu'une seule entité A sans récupérer les entités liées. Maintenant, voyons comment réaliser le deuxième cas, c'est-à-dire récupérer tout d'un coup avec une jointure, pour éviter une seconde requête par la suite.

Tout d'abord, rappelons le cas d'utilisation principal de ces jointures. C'est surtout lorsque vous bouclez sur une liste d'entités A (par exemple des articles), et que dans cette boucle vous faites `$entiteA->getEntiteB()` (par exemple des commentaires). Avec 1 requête par itération dans la boucle, vous explosez votre nombre de requêtes sur une seule page ! C'est donc principalement pour éviter cela que nous allons faire des jointures.

Comment faire des jointures avec le QueryBuilder ?

Heureusement, c'est très simple ! Voici tout de suite un exemple :

Code : PHP

```
<?php
// Depuis le repository d'Article
public function getArticleAvecCommentaires()
{
    $qb = $this->createQueryBuilder('a')
        ->leftJoin('a.commentaires', 'c')
        ->addSelect('c');

    return $qb->getQuery()
        ->getResult();
}
```

L'idée est donc très simple :

- D'abord on crée une jointure avec la méthode `leftJoin()` (ou `join()` pour faire l'équivalent d'un INNER JOIN). Le premier argument de la méthode est l'attribut de l'entité principale (celle qui est dans le `FROM` de la requête) sur lequel faire la jointure. Dans l'exemple, l'entité Article possède un attribut `commentaires`. Le deuxième argument de la méthode est l'alias de l'entité jointe.
- Puis on sélectionne également l'entité jointe, via un `addSelect()`. En effet un `select()` tout court aurait écrasé le `select ('a')` déjà fait par le `createQueryBuilder()`, rappelez-vous.

Attention : On ne peut faire une jointure que si l'entité du `FROM` possède un attribut vers l'entité à joindre ! Cela veut dire que soit l'entité du `FROM` est l'entité propriétaire de la relation, soit la relation est bidirectionnelle.

Dans notre exemple, la relation entre Article et Commentaire est une ManyToOne avec Commentaire le côté Many, le côté propriétaire donc. Cela veut dire que pour pouvoir faire la jointure dans ce sens, la relation est bidirectionnelle, afin d'ajouter un attribut "commentaires" dans l'entité inverse Article.



Et pourquoi on n'a pas précisé la condition ON du JOIN ?

C'est une bonne question. La réponse est très logique, pour cela réfléchissez plutôt à la question suivante : pourquoi est-ce qu'on rajoute un `ON` habituellement dans nos requêtes SQL ? C'est pour que MySQL (ou tout autre SGBDR) puisse savoir sur quelle condition faire la jointure. Or ici, on s'adresse à Doctrine et non directement à MySQL. Et bien entendu, Doctrine connaît déjà tout sur notre association, grâce aux annotations !

Bien sûr, vous pouvez toujours personnaliser la condition de jointure, en rajoutant vos conditions à la suite du `ON` généré par Doctrine, grâce à la syntaxe du `WITH` :

Code : PHP

```
<?php
$qb->join('a.commentaires', 'c', 'WITH', 'YEAR(c.date) > 2011')
```

Le troisième argument est le type de condition `WITH`, et le 4^e argument est ladite condition.



WITH ? C'est quoi cette syntaxe pour faire une jointure ?

En SQL, la différence entre le **ON** et le **WITH** est simple : un **ON** définit la condition pour la jointure, alors qu'un **WITH** ajoute une condition pour la jointure. Attention, en DQL le **ON** n'existe pas, seul le **WITH** est supporté. Ainsi, la syntaxe précédente avec le **WITH** serait équivalente à la syntaxe SQL suivante à base de **ON** :

Code : SQL

```
SELECT * FROM Article a JOIN Commentaire c ON c.article = a.id AND
YEAR(c.date) > 2011
```

Grâce au **WITH**, on n'a pas besoin de réécrire la condition par défaut de la jointure, le **c.article = a.id**.

Comment utiliser les jointures ?

Réponse : comme d'habitude ! Vous n'avez rien à modifier dans votre code. Si vous utilisez une entité dont vous avez récupéré les entités liées avec une jointure, vous pouvez alors utiliser les getters joyeusement sans craindre de requête supplémentaire. Reprenons l'exemple de la méthode `getArticleAvecCommentaires()` définie précédemment, on pourrait utiliser les résultats comme ceci :

Code : PHP

```
<?php
// Depuis un contrôleur
public function listeAction()
{
    $liste_articles = $this->getDoctrine()
        ->getEntityManager()
        ->getRepository('SdzBlogBundle:Article')
        ->getArticleAvecCommentaires();

    foreach($liste_articles as $article)
    {
        $article->getCommentaires(); // Ne déclenche pas de requête :
        // les commentaires sont déjà chargés ! Vous pourriez faire une boucle
        // dessus pour les afficher tous.
    }
    // ...
}
```

Voici donc comment vous devrez faire la plupart de vos requêtes. En effet, vous aurez souvent besoin d'utiliser des entités liées entre elles, et faire une ou plusieurs jointures s'impose très souvent. 😊

Application : les entités de notre blog

Plan d'attaque

Nous allons ajouter une méthode dans l'`ArticleRepository` pour récupérer tous les articles qui correspondent à une liste de catégories. La définition de la méthode est donc `<?php getAvecCategories(array $nom_categories) ?>`, que l'on pourra utiliser comme cela par exemple : `<?php $articleRepository->getAvecCategories(array('Doctrine2', 'Tutorial')) ?>`.

À vous de jouer !



Important : faites-le vous-même ! La correction est juste en dessous, je sais, mais si vous ne faites pas maintenant l'effort d'y réfléchir par vous-même, cela vous handicadera par la suite !

Le code

`ArticleRepository.php` :
Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\EntityRepository;

/**
 * ArticleRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom

```

```
* repository methods below.
*/
class ArticleRepository extends EntityRepository
{
    public function getAvecCategories(array $nom_categories)
    {
        $qb = $this->createQueryBuilder('a');

        // On fait une jointure avec l'entité Categorie, avec pour
        alias « c »
        $qb ->join('a.categories', 'c')
            ->where($qb->expr()->in('c.nom', $nom_categories)); // Puis
        on filtre sur le nom des catégories à l'aide d'un IN

        // Enfin, on retourne le résultat
        return $qb->getQuery()
            ->getResult();
    }
}
```



Que faire avec ce que retourne cette fonction ?

Comme je l'ai précédemment, cette fonction va retourner un tableau d'Article. Qu'est-ce que l'on veut en faire ? Les afficher. Donc la première chose à faire est de passer ce tableau à Twig. Ensuite, dans Twig, vous faites un simple `{% for %}` pour afficher les articles. Ça n'est vraiment pas compliqué à utiliser !

Et voilà, vous avez tout le code. Je n'ai qu'une chose à vous dire à ce stade du cours : entraînez-vous ! Amusez-vous à faire des requêtes dans tous les sens dans l'ArticleRepository ou même dans les autres repository. Jouez avec les relations entre les entités, créez-en d'autres. Bref, ça ne viendra pas tout seul, il va falloir travailler un peu de votre côté. 😊

En résumé

- Le rôle d'un repository est, à l'aide du langage DQL ou du constructeur de requête, de récupérer des entités selon des contraintes, des tris, etc. ;
- Un repository dispose toujours de quelques méthodes de base, permettant de récupérer de façon très simple les entités ;
- Mais la plupart du temps, il faut créer des méthodes personnelles pour récupérer les entités exactement comme on le veut ;
- Il est indispensable de faire les bonnes jointures afin de limiter au maximum le nombre de requêtes SQL sur vos pages.

Les évènements et extensions Doctrine

Maintenant que vous savez manipuler vos entités, vous allez vous rendre compte que pas mal de comportements sont répétitifs. En bon développeur, il est hors de question de dupliquer du code ou de perdre du temps : nous sommes bien trop fainéants !

Ce chapitre a pour objectif de vous présenter les évènements et les extensions Doctrine, qui vous permettront de simplifier certains cas usuels que vous rencontrerez.

Les évènements Doctrine

L'intérêt des évènements Doctrine

Dans certains cas, vous pouvez avoir besoin d'effectuer des actions juste avant ou juste après qu'une entité ait été créée, mise à jour ou supprimée. Par exemple, si vous stockez la date d'édition d'un article, à chaque modification de l'entité Article il faut mettre à jour cet attribut juste avant la mise à jour dans la base de données.

Ces actions, vous devez les faire à chaque fois. Cet aspect systématique a deux impacts. D'une part, cela veut dire qu'il faut être sûr de vraiment les effectuer à chaque fois pour que votre base de données soit cohérente. D'autre part, cela veut dire qu'on est bien trop fainéant pour se répéter !

C'est ici qu'interviennent les évènements Doctrine. Plus précisément, vous les trouverez sous le nom de *callbacks* du cycle de vie (*lifecycle* en anglais) d'une entité. Un *callback* est une méthode de votre entité, et on va dire à Doctrine de l'exécuter à certains moments.

On parle d'évènements de « cycle de vie » car ce sont différents évènements que Doctrine lève à chaque moment de la vie d'une entité : son chargement depuis la base de données, sa modification, sa suppression, etc. On en reparle plus loin, je vous dresserai une liste complète des évènements et de leur utilisation.

Définir des callbacks de cycle de vie

Pour vous expliquer le principe, nous allons prendre l'exemple de notre entité Article, qui va comprendre un attribut \$dateEdition représentant la date de la dernière édition de l'article. Si vous ne l'avez pas déjà, ajoutez-le maintenant, et n'oubliez pas de mettre à jour la base de données à l'aide de la commande doctrine:schema:update.

1. Définir l'entité comme contenant des callbacks

Tout d'abord, on doit dire à Doctrine que notre entité contient des callbacks de cycle de vie ; cela se définit grâce à l'annotation HasLifecycleCallbacks dans le namespace habituel des annotations Doctrine :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Article
{
    // ...
}
```

Cette annotation permet à Doctrine de vérifier les callbacks éventuels contenus dans l'entité. Ne l'oubliez pas car sinon vos différents callbacks seront tout simplement ignorés.

2. Définir un callback et ses évènements associés

Maintenant, il faut définir des méthodes et surtout, les évènements sur lesquels elles seront exécutées.

Continuons dans notre exemple, et créons une méthode updateDate() dans l'entité Article. Cette méthode doit définir l'attribut \$dateEdition à la date actuelle, afin de mettre à jour automatiquement la date d'édition d'un article. Voici à quoi elle pourrait ressembler :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

/**
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Article
{
    // ...
}
```

```
/*
class Article
{
    // ...

    public function updateDate()
    {
        $this->setDateEdition(new \Datetime());
    }
}
```

Maintenant il faut dire à Doctrine d'exécuter cette méthode (ce *callback*) dès que l'entité `Article` est modifiée. On parle d'*écouter un évènement*. Il existe plusieurs évènements de cycle de vie avec Doctrine, celui qui nous intéresse ici est l'évènement `preUpdate` : c'est-à-dire que la méthode va être exécutée juste avant que l'entité ne soit modifiée en base de données. Voici à quoi cela ressemble :

Code : PHP

```
<?php

/**
 * @ORM\PreUpdate
 */
public function updateDate()
```

C'est tout !

Vous pouvez dès à présent tester le comportement. Essayez de faire un petit code de test pour charger un article, le modifier, et l'enregistrer (avec un `flush()`), vous verrez que l'attribut `$dateEdition` va se mettre à jour automatiquement. Attention l'évènement `update` n'est pas déclenché à la création d'une entité, mais seulement à sa modification : c'est parfaitement ce qu'on veut dans notre exemple.

Pour aller plus loin, il y a deux points qu'il vous faut savoir. D'une part, au même titre que l'évènement `preUpdate`, il existe l'évènement `postUpdate` et bien d'autres, on en dresse une liste dans le tableau suivant. D'autre part, vous l'avez sûrement noté mais le *callback* ne prend aucun argument, vous ne pouvez en effet utiliser et modifier que l'entité courante. Pour exécuter des actions plus complexes lors d'évènements, il faut créer des services, on voit ça plus loin.

Liste des évènements de cycle de vie

Les différents évènements du cycle de vie sont récapitulés dans le tableau suivant.

Meta-événement	Évènement	Description
Persist	PrePersist	L'évènement <code>prePersist</code> se produit juste avant que l'EntityManager ne persiste effectivement l'entité. Concrètement, cela exécute le <i>callback</i> juste avant un <code>\$em->persist(\$entity)</code> . Il ne concerne que les entités nouvellement créées. Du coup il y a deux conséquences : d'une part, les modifications que vous apportez à l'entité seront persistées en base de données, puisqu'elles sont effectives avant que l'EntityManager n'enregistre l'entité en base. D'autre part, vous n'avez pas accès à l'id de l'entité si celui-ci est auto-généré, car justement l'entité n'est pas encore enregistrée en base de données, et donc l'id pas encore généré.
	PostPersist	L'évènement <code>postPersist</code> se produit juste après que l'EntityManager ait effectivement persisté l'entité. Attention cela n'exécute pas le <i>callback</i> juste après le <code>\$em->persist(\$entity)</code> , mais juste après le <code>\$em->flush()</code> . À l'inverse du <code>prePersist</code> , les modifications que vous apportez à l'entité ne seront pas persistées en base (mais seront tout de même appliquées à l'entité attention) ; mais vous avez par contre accès à l'id qui a été généré lors du <code>flush()</code> .
Update	PreUpdate	L'évènement <code>preUpdate</code> se produit juste avant que l'EntityManager ne modifie une entité. Par modifiée, j'entends que l'entité existait déjà, que vous y avez apporté des modifications puis un <code>\$em->flush()</code> . Le <i>callback</i> sera exécuté juste avant le <code>flush()</code> . Attention il faut que vous ayez modifié au moins un attribut pour que l'Entity Manager génère une requête et donc déclenche cet évènement. Vous avez accès à l'id auto-généré (car l'entité existe déjà), et vos modifications seront persistées en base de données.
	PostUpdate	L'évènement <code>postUpdate</code> se produit juste après que l'EntityManager ait effectivement modifié une entité. Vous avez accès à l'id et vos modifications ne sont pas persistées en base de données.
Remove	PreRemove	L'évènement <code>preRemove</code> se produit juste avant que l'EntityManager ne supprime une entité, c'est-à-dire juste avant un <code>\$em->flush()</code> qui précède un <code>\$em->remove(\$entity)</code> . Attention soyez prudent dans cet évènement, si vous souhaitez supprimer des fichiers liés à l'entité par exemple, car à ce moment l'entité n'est pas encore effectivement supprimée, et la suppression peut être annulée en cas d'erreur dans une des opérations à effectuer dans le <code>flush()</code> .

	PostRemove	L'évènement PostRemove se produit juste après que l'EntityManager ait effectivement supprimé une entité. Si vous n'avez plus accès à son id, c'est ici que vous pouvez effectuer une suppression de fichier associé par exemple.
Load	PostLoad	L'évènement PostLoad se produit juste après que l'EntityManager ait chargé une entité (ou après un \$em->refresh()). Utile pour appliquer une action lors du chargement d'une entité.



Attention, ces évènements se produisent lorsque vous créez et modifiez vos entités en manipulant les objets. Ils ne sont pas déclenchés lorsque vous effectuez des requêtes DQL ou avec le *QueryBuilder*. Car ces requêtes peuvent toucher un grand nombre d'entités et il serait dangereux pour Doctrine de déclencher les évènements correspondants un à un.

Les extensions Doctrine

L'intérêt des extensions Doctrine

Dans la gestion des entités d'un projet, il y a des comportements assez communs que vous souhaiterez implémenter. Par exemple, il est très classique de vouloir générer des slugs pour les articles d'un blog, les sujets d'un forum, etc. Plutôt que de réinventer tout le comportement nous-mêmes, nous allons utiliser les extensions Doctrine !

Doctrine2 est en effet très flexible, et la communauté a déjà créé une série d'extensions Doctrine très pratiques afin de vous aider avec les tâches usuelles liées aux entités. À l'image des évènements, utiliser ces extensions évite de se répéter au sein de votre application Symfony2 : c'est la philosophie [DRY](#).

Installer le StofDoctrineExtensionBundle

Un bundle en particulier permet d'intégrer différentes extensions Doctrine dans un projet Symfony2, il s'agit de [StofDoctrineExtensionBundle](#). Commençons par l'installer avec Composer, rajoutez cette dépendance dans votre `composer.json` :

Code : JavaScript

```
// composer.json
"require": {
    "stof/doctrine-extensions-bundle": "dev-master"
}
```

Ce bundle intègre la bibliothèque [DoctrineExtensions](#) sous-jacente, qui est celle qui inclut réellement les extensions.

N'oubliez pas d'enregistrer le bundle dans le noyau :

Code : PHP

```
<?php
// app/AppKernel.php

public function registerBundles()
{
    return array(
        // ...
        new
Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
        // ...
    );
}
```

Voilà le bundle est installé, il faut maintenant activer telle ou telle extension.

Utiliser une extension : l'exemple de Sluggable

L'utilisation des différentes extensions est très simple grâce à la flexibilité de Doctrine2 et au bundle pour Symfony2. Voici par exemple l'utilisation de l'extension Sluggable, qui permet de définir très facilement un attribut `slug` dans une entité : le slug sera automatiquement généré !

Tout d'abord, il faut activer l'extension Sluggable, il faut pour cela configurer le bundle *via* le fichier de configuration `config.yml`. Rajoutez donc cette section :

Code : YAML

```
# app/config/config.yml
# Stof\DoctrineExtensionBundle configuration
stof doctrine_extensions:
```

```
    orm:
      default:
        sluggable: true
```

Cela va activer l'extension sluggable. De la même manière, vous pourrez activer les autres extensions en les rajoutant à la suite.

Concrètement, l'utilisation des extensions se fait grâce à de judicieuses annotations. Vous l'aurez deviné, pour l'extension Sluggable l'annotation est tout simplement Slug. En l'occurrence il faut ajouter un nouvel attribut slug (le nom est arbitraire) dans votre entité, sur lequel nous mettrons l'annotation. Voici un exemple dans notre entité Article :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Gedmo\Mapping\Annotation as Gedmo;

class Article
{
    // ...

    /**
     * @Gedmo\Slug(fields={"titre"})
     * @ORM\Column(length=128, unique=true)
     */
    private $slug;

    // ...
}
```

Dans un premier temps, vous avez l'habitude, on utilise le namespace de l'annotation, ici Gedmo\Mapping\Annotation.

Ensuite, l'annotation Slug s'utilise très simplement sur un attribut qui va contenir le slug. L'option fields permet de définir le ou les attributs à partir desquels le slug sera généré : ici le titre uniquement. Mais vous pouvez en indiquer plusieurs en les séparant par des virgules.



N'oubliez pas de mettre à jour votre base de données avec la commande doctrine:schema:update, mais également de générer le getter et le setter du slug, grâce à la commande generate:doctrine:entities SdzBlogBundle:Article.

C'est tout ! Vous pouvez dès à présent tester le nouveau comportement de votre entité. Créez une entité avec un titre de test, et enregistrez-la : son attribut slug sera automatiquement rempli ! Par exemple :

Code : PHP

```
<?php
// Dans un contrôleur

public function testAction()
{
    $article = new Article();
    $article->setTitre("L'histoire d'un bon weekend !");

    $em = $this->getDoctrine()->getManager();
    $em->persist($article);
    $em->flush(); // C'est à ce moment qu'est généré le slug

    return new Response('Slug généré : '.$article->getSlug()); // Affiche << Slug généré : l-histoire-d-un-bon-weekend>>
}
```

L'attribut slug est rempli automatiquement par le bundle. Ce dernier utilise en réalité tout simplement les événements Doctrine PrePersist et PreUpdate, qui permettent d'intervenir juste avant l'enregistrement et la modification de l'entité comme on l'a vu plus haut.



Vous avez pu remarquer que j'ai défini l'attribut slug comme unique (unique=true dans l'annotation Column). En effet dans le cadre d'un article de blog, on se sert souvent du slug comme identifiant de l'article, afin de l'utiliser dans les URL et améliorer le référencement. Sachez que l'extension est intelligente : si vous ajoutez un Article avec un titre qui existe déjà, le slug sera suffixé de « -1 » pour garder l'unicité, par exemple "un-super-weekend-1". Si vous ajoutez un troisième titre identique, alors le slug sera « un-super-weekend-2 », etc.

Vous savez maintenant utiliser l'extension Doctrine Sluggable ! Voyons les autres extensions disponibles.

Liste des extensions Doctrine

Voici la liste de toutes les extensions actuellement disponibles, ainsi que leur description et des liens vers la documentation pour vous permettre de les implémenter dans votre projet.

Extension	Description
Tree	L'extension Tree automatise la gestion des arbres et ajoute des méthodes spécifiques au repository. Les arbres sont une représentation d'entités avec des liens type parents-enfants, utiles pour les catégories d'un forum par exemple.
Translatable	L'extension Translatable offre une solution aisée pour traduire des attributs spécifiques de vos entités dans différents langages. De plus, elle charge automatiquement les traductions pour la locale courante.
Sluggable	L'extension Sluggable permet de générer automatiquement un slug à partir d'attributs spécifiés.
Timestampable	L'extension Timestampable automatise la mise à jour d'attribut de type date dans vos entités. Vous pouvez définir la mise à jour d'un attribut à la création et/ou à la modification, ou même à la modification d'un attribut particulier. Vous l'aurez compris, cette extension fait la même chose que ce qu'on a fait dans le paragraphe précédent sur les événements Doctrine, et en mieux !
Loggable	L'extension Loggable permet de conserver les différentes versions de vos entités, et offre des outils de gestion des versions.
Sortable	L'extension Sortable permet de gérer des entités ordonnées, c'est-à-dire avec un ordre précis.
Softdeleteable	L'extension SoftDeleteable permet de « soft-supprimer » des entités, c'est-à-dire de ne pas les supprimer réellement, juste mettre un de leur attribut à <code>true</code> pour les différencier. L'extension permet également de les filtrer lors des SELECT, pour ne pas utiliser des entités « soft-supprimées ».
Uploadable	L'extension Uploadable offre des outils pour gérer l'enregistrement de fichiers associés avec des entités. Elles inclut la gestion automatique des déplacements et de suppressions des fichiers.

Si vous n'avez pas besoin aujourd'hui de tous ces comportements, ayez-les en tête pour le jour où vous en trouverez l'utilité. Autant ne pas réinventer la roue si elle existe déjà ! 😊

Pour conclure

Ce chapitre touche à sa fin et marque la fin de la partie théorique sur Doctrine. Vous avez maintenant tous les outils pour gérer vos entités, et donc votre base de données. Surtout, n'hésitez pas à bien pratiquer, car c'est une partie qui implique de nombreuses notions : sans entraînement, pas de succès !

Le prochain chapitre est un TP permettant de mettre en pratique la plupart des notions abordées dans cette partie.

En résumé

- Les événements permettent de centraliser du code répétitif, afin de systématiser leur exécution et de réduire la duplication de code ;
- Plusieurs événements jalonnent la vie d'une entité, afin de pouvoir exécuter une fonction aux endroits désirés ;
- Les extensions permettent de reproduire des comportements communs dans une application, afin d'éviter de réinventer la roue.

TP : Les entités de notre blog

L'objectif de ce chapitre est de mettre en application tout ce que nous avons vu au cours de cette partie sur Doctrine2. Nous allons créer les entités Article et Blog, mais également adapter le contrôleur pour nous en servir. Enfin, nous verrons quelques astuces de développement Symfony2 au cours du TP.

Surtout, je vous invite à bien essayer de réfléchir par vous même avant de lire les codes que je donne. C'est ce mécanisme de recherche qui va vous faire progresser sur Symfony2, il serait dommage de s'en passer !

Bon TP !

Synthèse des entités

Entité Article

On a déjà pas mal traité l'entité Article au cours de cette partie. Pour l'instant, on a toujours le pseudo de l'auteur écrit en dur dans l'entité. Souvenez-vous, pour commencer, on n'a pas d'entité Utilisateur, on doit donc écrire le pseudo de l'auteur en dur dans les articles.

Voici donc la version finale de l'entité Article que vous devriez avoir :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Gedmo\Mapping\Annotation as Gedmo;

/**
 * Sdz\BlogBundle\Entity\Article
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Article
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var datetime $date
     *
     * @ORM\Column(name="date", type="datetime")
     */
    private $date;

    /**
     * @var string $titre
     *
     * @ORM\Column(name="titre", type="string", length=255)
     */
    private $titre;

    /**
     * @var string $auteur
     *
     * @ORM\Column(name="auteur", type="string", length=255)
     */
    private $auteur;

    /**
     * @var boolean $publication
     *
     * @ORM\Column(name="publication", type="boolean")
     */
    private $publication;

    /**
     * @var text $contenu
     *
     * @ORM\Column(name="contenu", type="text")
     */
    private $contenu;

    /**
     * @ORM\Column(type="date", nullable=true)
     */
    private $dateEdition;
```

```

    /**
 * @Gedmo\Slug(fields={"titre"})
 * @ORM\Column(length=128, unique=true)
 */
private $slug;

    /**
 * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image",
cascade={"persist"})
*/
private $image;

    /**
 * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Categorie",
cascade={"persist"})
*/
private $categories;

    /**
 * @ORM\OneToMany(targetEntity="Sdz\BlogBundle\Entity\Commentaire",
mappedBy="article")
*/
private $commentaires; // Ici commentaires prend un « s », car un
article a plusieurs commentaires !

public function __construct()
{
    $this->date      = new \Datetime;
    $this->publication = true;
    $this->categories = new
\Doctrine\Common\Collections\ArrayCollection();
    $this->commentaires = new
\Doctrine\Common\Collections\ArrayCollection();
}

/**
 * @ORM\preUpdate
 * Callback pour mettre à jour la date d'édition à chaque
modification de l'entité
*/
public function updateDate()
{
    $this->setDateEdition(new \Datetime());
}

/**
 * @return integer
*/
public function getId()
{
    return $this->id;
}

/**
 * @param datetime $date
 * @return Article
*/
public function setDate(\Datetime $date)
{
    $this->date = $date;
    return $this;
}

/**
 * @return datetime
*/
public function getDate()
{
    return $this->date;
}

/**
 * @param string $titre
 * @return Article
*/
public function setTitre($titre)
{
    $this->titre = $titre;
    return $this;
}

/**
 * @return string
*/
public function getTitre()
{
    return $this->titre;
}

```

```
    /**
 * @param text $contenu
 * @return Article
 */
    public function setContenu($contenu)
{
    $this->contenu = $contenu;
    return $this;
}

    /**
 * @return text
 */
    public function getContenu()
{
    return $this->contenu;
}

    /**
 * @param boolean $publication
 * @return Article
 */
    public function setPublication($publication)
{
    $this->publication = $publication;
    return $this;
}

    /**
 * @return boolean
 */
    public function getPublication()
{
    return $this->publication;
}

    /**
 * @param string $auteur
 * @return Article
 */
    public function setAuteur($auteur)
{
    $this->auteur = $auteur;
    return $this;
}

    /**
 * @return string
 */
    public function getAuteur()
{
    return $this->auteur;
}

    /**
 * @param Sdz\BlogBundle\Entity\Image $image
 * @return Article
 */
    public function setImage(\Sdz\BlogBundle\Entity\Image $image = null)
{
    $this->image = $image;
    return $this;
}

    /**
 * @return Sdz\BlogBundle\Entity\Image
 */
    public function getImage()
{
    return $this->image;
}

    /**
 * @param Sdz\BlogBundle\Entity\Categorie $categorie
 * @return Article
 */
    public function addCategorie(\Sdz\BlogBundle\Entity\Categorie
$categorie)
{
    $this->categories[] = $categorie;
    return $this;
}

    /**
 * @param Sdz\BlogBundle\Entity\Categorie $categorie
 */
    public function removeCategorie(\Sdz\BlogBundle\Entity\Categorie
$categorie)
```

```

    {
        $this->categories->removeElement($categorie);
    }

    /**
 * @return Doctrine\Common\Collections\Collection
 */
    public function getCategories()
    {
        return $this->categories;
    }

    /**
 * @param Sdz\BlogBundle\Entity\Commentaire $commentaire
 * @return Article
 */
    public function addCommentaire (\Sdz\BlogBundle\Entity\Commentaire
$commentaire)
    {
        $this->commentaires[] = $commentaire;
        return $this;
    }

    /**
 * @param Sdz\BlogBundle\Entity\Commentaire $commentaire
 */
    public function removeCommentaire (\Sdz\BlogBundle\Entity\Commentaire
$commentaire)
    {
        $this->commentaires->removeElement($commentaire);
    }

    /**
 * @return Doctrine\Common\Collections\Collection
 */
    public function getCommentaires()
    {
        return $this->commentaires;
    }

    /**
 * @param datetime $dateEdition
 * @return Article
 */
    public function setDateEdition (\Datetime $dateEdition)
    {
        $this->dateEdition = $dateEdition;
        return $this;
    }

    /**
 * @return date
 */
    public function getDateEdition()
    {
        return $this->dateEdition;
    }

    /**
 * @param string $slug
 * @return Article
 */
    public function setSlug ($slug)
    {
        $this->slug = $slug;
        return $this;
    }

    /**
 * @return string
 */
    public function getSlug()
    {
        return $this->slug;
    }
}

```

Entité Image

L'entité `Image` est une entité très simple, qui nous servira par la suite pour l'upload d'images avec Symfony2. Sa particularité est qu'elle peut être liée à n'importe quelle autre entité : elle n'est pas du tout exclusive à l'entité `Article`. Si vous souhaitez ajouter des images ailleurs que dans des `Article`, il n'y aura aucun problème.

Voici son code, que vous devriez déjà avoir :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Image.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ImageRepository")
 */
class Image
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $url
     *
     * @ORM\Column(name="url", type="string", length=255)
     */
    private $url;

    /**
     * @var string $salt
     *
     * @ORM\Column(name="alt", type="string", length=255)
     */
    private $salt;

    /**
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @param string $url
     * @return Image
     */
    public function setUrl($url)
    {
        $this->url = $url;
        return $this;
    }

    /**
     * @return string
     */
    public function getUrl()
    {
        return $this->url;
    }

    /**
     * @param string $salt
     * @return Image
     */
    public function setAlt($salt)
    {
        $this->alt = $salt;
        return $this;
    }

    /**
     * @return string
     */
    public function getAlt()
    {
        return $this->alt;
    }
}

```

Entité Commentaire

L'entité Commentaire, bien que très simple, contient la relation avec l'entité Article, c'est elle la propriétaire. Voici son code :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
* @ORM\Table()
*
@ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CommentaireRepository")
*/
class Commentaire
{
    /**
* @ORM\Column(name="id", type="integer")
* @ORM\Id
* @ORM\GeneratedValue(strategy="AUTO")
*/
    private $id;

    /**
* @ORM\Column(name="auteur", type="string", length=255)
*/
    private $auteur;

    /**
* @ORM\Column(name="contenu", type="text")
*/
    private $contenu;

    /**
* @ORM\Column(name="date", type="datetime")
*/
    private $date;

    /**
* @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article",
inversedBy="commentaires")
* @ORM\JoinColumn(nullable=false)
*/
    private $article;

    public function __construct()
    {
        $this->date = new \Datetime();
    }

    /**
* @return integer
*/
    public function getId()
    {
        return $this->id;
    }

    /**
* @param string $auteur
* @return Commentaire
*/
    public function setAuteur($auteur)
    {
        $this->auteur = $auteur;
        return $this;
    }

    /**
* @return string
*/
    public function getAuteur()
    {
        return $this->auteur;
    }

    /**
* @param text $contenu
* @return Commentaire
*/
    public function setContenu($contenu)
    {
```

```

        $this->contenu = $contenu;
        return $this;
    }

    /**
 * @return text
 */
    public function getContenu()
    {
        return $this->contenu;
    }

    /**
 * @param datetime $date
 * @return Commentaire
 */
    public function setDate(\Datetime $date)
    {
        $this->date = $date;
        return $this;
    }

    /**
 * @return datetime
 */
    public function getDate()
    {
        return $this->date;
    }

    /**
 * @param Sdz\BlogBundle\Entity\Article $article
 * @return Commentaire
 */
    public function setArticle(\Sdz\BlogBundle\Entity\Article $article)
    {
        $this->article = $article;
        return $this;
    }

    /**
 * @return Sdz\BlogBundle\Entity\Article
 */
    public function getArticle()
    {
        return $this->article;
    }
}

```

Entité Categorie

L'entité Categorie ne contient qu'un attribut nom (enfin, vous pouvez en rajouter de votre côté bien sûr!). La relation avec Article est contenue dans l'entité Article, qui en est la propriétaire. Voici son code, que vous devriez déjà avoir :

Code : PHP

```

<?php
// src/Sdz/Bundle/Entity/Categorie.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table()
 */
@ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CategorieRepository")
class Categorie
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $nom
     *
     * @ORM\Column(name="nom", type="string", length=255)
     */
}

```

```

private $nom;

/**
* @return integer
*/
public function getId()
{
    return $this->id;
}

/**
* @param string $nom
* @return Categorie
*/
public function setNom($nom)
{
    $this->nom = $nom;
    return $this;
}

/**
* @return string
*/
public function getNom()
{
    return $this->nom;
}
}

```

Entités Competence et ArticleCompetence

L'entité Competence ne contient, au même titre que l'entité Categorie, qu'un attribut nom, mais vous pouvez bien sûr en rajouter selon vos besoins. Voici son code :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/Competence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
* @ORM\Table()
*
* @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\CompetenceRepository")
*/
class Competence
{
    /**
     * @var integer $id
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $nom
     *
     * @ORM\Column(name="nom", type="string", length=255)
     */
    private $nom;

    /**
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @param string $nom
     * @return Competence
     */
    public function setNom($nom)
    {
        $this->nom = $nom;
        return $this;
    }
}

```

```
    }

    /**
 * @return string
 */
public function getNom()
{
    return $this->nom;
}
}
```

L'entité ArticleCompetence est l'entité de relation entre Article et Competence. Elle contient les attributs \$article et \$competence qui permettent de faire la relation, ainsi que d'autres attributs pour caractériser la relation, ici j'ai utilisé un attribut niveau. Voici son code, vous pouvez bien entendu rajouter les attributs de relation que vous souhaitez :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleCompetence.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class ArticleCompetence
{
    /**
     * @ORM\Id
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Article")
     */
    private $article;

    /**
     * @ORM\Column()
     */
    private $niveau; // Ici j'ai un attribut de relation, que j'ai
appelé « niveau »

    /**
     * @param string $niveau
     * @return ArticleCompetence
     */
    public function setNiveau($niveau)
    {
        $this->niveau = $niveau;
        return $this;
    }

    /**
     * @return string
     */
    public function getNiveau()
    {
        return $this->niveau;
    }

    /**
     * @param Sdz\BlogBundle\Entity\Article $article
     * @return ArticleCompetence
     */
    public function setArticle(\Sdz\BlogBundle\Entity\Article
$article)
    {
        $this->article = $article;
        return $this;
    }

    /**
     * @return Sdz\BlogBundle\Entity\Article
     */
    public function getArticle()
    {
        return $this->article;
    }

    /**
     * @param Sdz\BlogBundle\Entity\Competence $competence
     * @return ArticleCompetence
     */
    public function setCompetence(\Sdz\BlogBundle\Entity\Competence
$competence)
    {
        $this->competence = $competence;
        return $this;
    }

    /**
     * @return Competence
     */
    public function getCompetence()
    {
        return $this->competence;
    }
}
```

```


    /**
     * @param Sdz\BlogBundle\Entity\Competence $competence
     */
    public function setCompetence(\Sdz\BlogBundle\Entity\Competence
$competence)
{
    $this->competence = $competence;
    return $this;
}

/**
 * @return Sdz\BlogBundle\Entity\Competence
 */
public function getCompetence()
{
    return $this->competence;
}


```

Et bien sûr...

Si vous avez ajouté et/ou modifié des entités, n'oubliez pas de mettre à jour votre base de données ! Vérifiez les requêtes avec `php app/console doctrine:schema:update --dump-sql` puis exécutez les avec `--force`.

Adaptation du contrôleur

Théorie

Maintenant que l'on a nos entités, on va enfin pouvoir adapter notre contrôleur Blog pour qu'il récupère et modifie des vrais articles dans la base de données, et non plus nos articles statiques définis à la va-vite.

Pour cela, il y a très peu de modifications à réaliser : voici encore un exemple du code découpé que Symfony2 nous permet de réaliser ! En effet, il vous suffit de modifier les quatre endroits où on avait écrit un article en dur dans le contrôleur. Modifiez ces quatre endroits en utilisant bien le repository de l'entité Article, seules les méthodes `findAll()` et `find()` vont nous servir pour le moment.

Attention je vous demande également de faire attention au cas où l'article demandé n'existe pas. Si on essaie d'aller à la page `/blog/article/4` alors que l'article d'id 4 n'existe pas, je veux une erreur correctement générée ! On a déjà vu le déclenchement d'une erreur 404 lorsque le paramètre `page` de la page d'accueil n'était pas valide, reprenez ce comportement.

À la fin le contrôleur ne sera pas entièrement opérationnel, car il nous manque toujours la gestion des formulaires. Mais il sera déjà mieux avancé !

Et bien sûr, n'hésitez pas à nettoyer tous les codes de tests qu'on a pu utiliser lors de cette partie pour jouer avec les entités, maintenant on doit avoir un vrai contrôleur qui ne fait que son rôle.

Pratique

Il n'y a vraiment rien de compliqué dans notre nouveau contrôleur, le voici :

Code : PHP

```


<?php

// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sdz\BlogBundle\Entity\Article;

class BlogController extends Controller
{
    public function indexAction($page)
    {
        // On ne sait pas combien de pages il y a
        // Mais on sait qu'une page doit être supérieure ou égale à 1
        // Bien sûr pour le moment on ne se sert pas (encore !) de
        // cette variable
        if ($page < 1) {
            // On déclenche une exception NotFoundHttpException
            // Cela va afficher la page d'erreur 404
            // On pourra la personnaliser plus tard
            throw $this->createNotFoundException('Page inexistante (page =
'. $page.'));
        }

        // Pour récupérer la liste de tous les articles : on utilise
        findAll()
        $articles = $this->getDoctrine()
            ->getManager()
            ->getRepository('SdzBlogBundle:Article')


```

```

        ->findAll();

    // L'appel de la vue ne change pas
    return $this->render('SdzBlogBundle:Blog:index.html.twig',
array(
    'articles' => $articles
));
}

public function voirAction($id)
{
    // On récupère l'EntityManager
    $em = $this->getDoctrine()
        ->getManager();

    // Pour récupérer un article unique : on utilise find()
    $article = $em->getRepository('SdzBlogBundle:Article')
        ->find($id);

    if ($article === null) {
        throw $this->createNotFoundException('Article[id='.$id.']'
inexistant.');
    }

    // On récupère les articleCompetence pour l'article $article
    $liste_articleCompetence = $em-
>getRepository('SdzBlogBundle:ArticleCompetence')
        ->findByArticle($article-
>getId());

    // Puis modifiez la ligne du render comme ceci, pour prendre en
    // compte les variables :
    return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
        'article'          => $article,
        'liste_articleCompetence' => $liste_articleCompetence,
        // Pas besoin de passer les commentaires à la vue, on pourra
        // y accéder via {{ article.commentaires }}
        // 'liste_commentaires' => $article->getCommentaires()
    ));
}

public function ajouterAction()
{
    // La gestion d'un formulaire est particulière, mais l'idée est
    // la suivante :

    if ($this->get('request')->getMethod() == 'POST') {
        // Ici, on s'occupera de la création et de la gestion du
        // formulaire

        $this->get('session')->setFlashBag()->add('info', 'Article
        bien enregistré');

        // Puis on redirige vers la page de visualisation de cet
        // article
        return $this->redirect( $this->generateUrl('sdzblog_voir',
array('id' => 1)) );
    }

    // Si on n'est pas en POST, alors on affiche le formulaire
    return $this->render('SdzBlogBundle:Blog:ajouter.html.twig');
}

public function modifierAction($id)
{
    // On récupère l'EntityManager
    $em = $this->getDoctrine()
        ->getManager();

    // On récupère l'entité correspondant à l'id $id
    $article = $em->getRepository('SdzBlogBundle:Article')
        ->find($id);

    // Si l'article n'existe pas, on affiche une erreur 404
    if ($article == null) {
        throw $this->createNotFoundException('Article[id='.$id.']'
inexistant.');
    }

    // Ici, on s'occupera de la création et de la gestion du
    // formulaire

    return $this->render('SdzBlogBundle:Blog:modifier.html.twig',
array(
    'article' => $article
));
}

public function supprimerAction($id)

```

```

{
    // On récupère l'EntityManager
    $em = $this->getDoctrine()
        ->getEntityManager();

    // On récupère l'entité correspondant à l'id $id
    $article = $em->getRepository('SdzBlogBundle:Article')
        ->find($id);

    // Si l'article n'existe pas, on affiche une erreur 404
    if ($article == null) {
        throw $this->createNotFoundException('Article[id='.$id.]'
inexistant');
    }

    if ($this->get('request')->getMethod() == 'POST') {
        // Si la requête est en POST, on supprimera l'article
        $this->get('session')->setFlashBag()->add('info', 'Article
bien supprimé');

        // Puis on redirige vers l'accueil
        return $this->redirect($this->generateUrl('sdzblog_accueil'))
    }
}

// Si la requête est en GET, on affiche une page de
confirmation avant de supprimer
return $this->render('SdzBlogBundle:Blog:supprimer.html.twig',
array(
    'article' => $article
));
}

public function menuAction($nombre)
{
    $liste = $this->getDoctrine()
        ->getManager()
        ->getRepository('SdzBlogBundle:Article')
        ->findBy(
            array(),
            // Pas de critère
            array('date' => 'desc'), // On tri par date
            // On sélectionne $nombre
            $nombre,
            // À partir du premier
            0
        );

    return $this->render('SdzBlogBundle:Blog:menu.html.twig', array(
        'liste_articles' => $liste // C'est ici tout l'intérêt : le
        contrôleur passe les variables nécessaires au template !
    ));
}
}

```

Amélioration du contrôleur

Le contrôleur qu'on vient de faire n'est pas encore parfait, on peut faire encore mieux !

Je pense notamment à deux points en particulier :

- Le premier est l'utilisation implicite d'un `ParamConverter` pour éviter de vérifier à chaque fois l'existence d'un article (l'erreur 404 qu'on déclenche le cas échéant);
- Le deuxième est la pagination des articles sur la page d'accueil.

L'utilisation d'un `ParamConverter`

Le `ParamConverter` est une notion très simple : c'est un convertisseur de paramètre. Vous voilà bien avancé n'est-ce pas ?



Plus sérieusement, c'est vraiment cela. Il existe [un chapitre dédié](#) sur l'utilisation du `ParamConverter` dans la partie astuces. Mais sachez qu'il va convertir le paramètre en entrée du contrôleur dans une autre forme. Typiquement, nous avons le paramètre `$id` en entrée de certaines de nos actions, que nous voulons transformer directement en entité `Article`. Pour cela rien de plus simple, il faut modifier la définition des méthodes du contrôleur comme suit :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// La définition :

public function voirAction($id)

```

```
// Devient :

public function voirAction(Article $article)
```



Bien entendu, n'oubliez pas de rajouter en début de fichier `use Sdz\BlogBundle\Entity\Article;` afin de pouvoir utiliser `Article`.

Et grâce aux mécanismes internes de Symfony2, vous retrouvez directement dans la variable `$article` l'article correspondant à l'id `$id`. Cela nous permet de supprimer l'utilisation du repository pour récupérer l'article, mais également le test de l'existence de l'article. Ces deux points sont fait automatiquement par le `ParamConverter` de Doctrine, et cela simplifie énormément nos méthodes. Voyez par vous-même ce que devient la méthode `voirAction` du contrôleur :

Code : PHP

```
<?php
// src/Sdz/Bundle/Controller/BlogController.php

public function voirAction(Article $article)
{
    // À ce stade, la variable $article contient une instance de la
    // classe Article
    // Avec l'id correspondant à l'id contenu dans la route !

    // On récupère ensuite les articleCompetence pour l'article
    $article
        // On doit le faire à la main pour l'instant car la relation
        est unidirectionnelle
        // C'est-à-dire que $article->getArticleCompetences() n'existe
        pas !
    $listeArticleCompetence = $this->getDoctrine()
        ->getManager()
        -
    >getRepository('SdzBlogBundle:ArticleCompetence')
        ->findByArticle($article-
    >getId());

    return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
        'article'          => $article,
        'listeArticleCompetence'  => $listeArticleCompetence
    ));
}
```

Merveilleux non ? La méthode est réduit à son strict minimum, sans rien enlever en termes de fonctionnalité. Essayez avec un article qui n'existe pas : [/blog/article/5123123](#) ; nous avons bien une erreur 404.

Utiliser une jointure pour récupérer les articles

Actuellement sur la page d'accueil, avec l'action `indexAction()`, on ne récupère que les articles en eux-mêmes. Comme on en a parlé dans les précédents chapitres, cela veut dire que dans la boucle pour afficher les articles, on ne peut pas utiliser les informations sur les relations (dans notre cas, les attributs `$image`, `$categories` et `$commentaires`). Enfin, on peut bien entendu les utiliser via `$article->getImage()`, mais dans ce cas une requête sera générée pour aller récupérée l'image... à chaque itération de la boucle sur les articles !

Ce comportement est bien sûr à proscrire, car le nombre de requêtes SQL va monter en flèche et ce n'est pas bon du tout pour les performances. Il faut donc modifier la requête initiale qui récupère les articles, pour y rajouter des jointures qui vont récupérer en une seule requête les articles ainsi que leurs entités jointes.

Tout d'abord, on va créer une méthode `getArticles()` dans le repository de l'entité `Article`, une version toute simple qui ne fait que récupérer les entités ordonnées :

Code : PHP

```
<?php
// src/Sdz/Bundle/Entity/ArticleRepository.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ArticleRepository extends EntityRepository
{
    public function getArticles()
    {
        $query = $this->createQueryBuilder('a')
            ->orderBy('a.date', 'DESC')
            ->getQuery();
    }
}
```

```

        return $query->getResult();
    }
}

```

Adaptons ensuite le contrôleur pour utiliser cette nouvelle méthode :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

class BlogController extends Controller
{
    public function indexAction($page)
    {
        // On ne sait pas combien de pages il y a
        // Mais on sait qu'une page doit être supérieure ou égale à 1
        // Bien sûr pour le moment on ne se sert pas (encore !) de
        // cette variable
        if ($page < 1) {
            // On déclenche une exception NotFoundHttpException
            // Cela va afficher la page d'erreur 404
            // On pourra la personnaliser plus tard
            throw $this->createNotFoundException('Page inexisteante (page = '.
                '$page.)');
        }

        // Pour récupérer la liste de tous les articles : on utilise
        // notre nouvelle méthode
        $articles = $this->getDoctrine()
            ->getManager()
            ->getRepository('SdzBlogBundle:Article')
            ->getArticles();

        // L'appel de la vue ne change pas
        return $this->render('SdzBlogBundle:Blog:index.html.twig',
            array(
                'articles' => $articles
            ));
    }
}

```

Maintenant, il nous faut mettre en place les jointures dans la méthode `getArticles()`, afin de charger toutes les informations sur les articles et éviter les dizaines de requêtes supplémentaires.

Dans notre exemple, nous allons afficher les données de l'entité `image` et des entités catégories liées à chaque article. Il nous faut donc rajouter les jointures sur ces deux entités. On a déjà vu comment faire ces jointures, n'hésitez pas à essayer de les faire de votre côté avant de regarder ce code :

Code : PHP

```

<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

// ...

public function getArticles($nombreParPage, $page)
{
    $query = $this->createQueryBuilder('a')
        // On joint sur l'attribut « image »
        ->leftJoin('a.image', 'i')
        ->addSelect('i')
        // On joint sur l'attribut « categories »
        ->leftJoin('a.categories', 'c')
        ->addSelect('c')
        ->orderBy('a.date', 'DESC')
        ->getQuery();

    return $query->getResult();
}

```

Comme vous pouvez le voir, les jointures se font simplement en utilisant les attributs existants de l'entité racine, ici l'entité `Article`. On rajoute donc juste les `leftJoin()` et les `addSelect()`, afin que Doctrine n'oublie pas de sélectionner les données qu'il joint. C'est tout ! Vous pouvez maintenant utiliser un `$article->getImage()` sans déclencher de nouvelle requête.

Soit dit en passant, ces jointures peuvent justifier la mise en place d'une relation bidirectionnelle. En effet, dans l'état



actuel on ne peut pas récupérer les informations des compétences liées à un article par exemple, car l'entité Article n'a pas d'attribut ArticleCompetences, donc pas de ->leftJoin() possible. C'est l'entité ArticleCompetence qui est propriétaire de la relation unidirectionnelle. Si vous voulez afficher les compétences, vous devez commencer par rendre la relation bidirectionnelle. N'hésitez pas à le faire, c'est un bon entraînement !

La pagination des articles sur la page d'accueil

Paginer manuellement les résultats d'une requête n'est pas trop compliqué, il faut juste faire un peu de mathématiques à l'aide des variables suivantes :

- Nombre total d'articles ;
- Nombre d'articles à afficher par page ;
- La page courante.

Cependant, c'est un comportement assez classique et en bon développeur que nous sommes, trouvons une méthode plus simple et déjà prête ! Il existe en effet un paginateur intégré dans Doctrine2, qui permet de faire tout ça très simplement. Intégrons-le dans notre méthode getArticles() comme ceci :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\EntityRepository;
use Doctrine\ORM\Tools\Pagination\Paginator;

class ArticleRepository extends EntityRepository
{
    // On ajoute 2 arguments : le nombre d'article par page, ainsi
    // que la page courante
    public function getArticles($nombreParPage, $page)
    {
        // On déplace la vérification du numéro de page dans cette
        // méthode
        if ($page < 1) {
            throw new \InvalidArgumentException('L\'argument $page ne peut
            être inférieur à 1 (valeur : "'.$page.'")');
        }

        // La construction de la requête reste inchangée
        $query = $this->createQueryBuilder('a')
            ->leftJoin('a.image', 'i')
            ->addSelect('i')
            ->leftJoin('a.categories', 'cat')
            ->addSelect('cat')
            ->orderBy('a.date', 'DESC')
            ->getQuery();

        // On définit l'article à partir duquel commencer la liste
        $query->setFirstResult(($page-1) * $nombreParPage)
        // Ainsi que le nombre d'articles à afficher
        ->setMaxResults($nombreParPage);

        // Enfin, on retourne l'objet Paginator correspondant à la
        // requête construite
        // (n'oubliez pas le use correspondant en début de fichier)
        return new Paginator($query);
    }
}
```

Maintenant que cette méthode est fonctionnelle pour la pagination, je vous invite à adapter le contrôleur pour prendre en compte cette pagination. Il faut donc utiliser correctement la méthode du repository que l'on vient de détailler, mais également donner à la vue toutes les données dont elle a besoin pour afficher une liste des pages existantes.



Au travail ! Encore une fois, faites l'effort de la réaliser de votre côté. Évidemment je vous donne la solution, mais si vous n'avez pas essayé de chercher, vous ne progresserez pas. Courage !

Il existe bien entendu différentes manières de le faire, mais voici le code du contrôleur que je vous propose :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

public function indexAction($page)
{
```

```

    $articles = $this->getDoctrine()
        ->getManager()
        ->getRepository('SdzBlogBundle:Article')
        ->getArticles(3, $page); // 3 articles par page
    : c'est totalement arbitraire !

    // On ajoute ici les variables page et nb_page à la vue
    return $this->render('SdzBlogBundle:Blog:index.html.twig',
array(
    'articles' => $articles,
    'page'      => $page,
    'nombrePage' => ceil(count($articles)/3)
));
}

```

C'est tout ! En effet rappelez-vous l'architecture MVC, toute la logique de récupération des données est dans la couche modèle : ici notre repository. Notre contrôleur est donc réduit à son strict minimum, la couche modèle grâce à un Doctrine2 généreux en fonctionnalités fait tout le travail.

 Attention à une petite subtilité. Ici, la variable `$articles` contient une instance de `Paginator`. Concrètement, c'est une liste d'articles, dans notre cas une liste de 3 articles (on a mis cette valeur en dur). Vous pouvez l'utiliser avec un simple `foreach` par exemple. Cependant, pour obtenir le nombre de pages vous voyez qu'on a utilisé un `count ($articles)` : ce `count` ne retourne pas 3, mais le nombre total d'articles dans la base de données ! Ceci est possible avec les objets qui implémentent l'interface `Countable` de PHP.

Enfin, il nous reste juste la vue à adapter. Voici ce que je peux vous proposer, j'ai juste rajouté l'affichage de la liste des pages possibles :

Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #-}

{# ... le reste de la vue #-}

<div class="pagination">
<ul>
    {# On utilise la fonction range(a, b) qui crée un tableau de
    valeurs entre a et b #}
    {%
        for p in range(1, nombrePage) %
    %}
        {%
            if p == page %} class="active"{% endif %}>
            <a href="{{ path('sdzblog_accueil', {'page': p}) }}">{{ p
        }}</a>
        {%
            endfor %
    %}
</ul>
</div>

```

Ce qui donne le résultat visible à la figure suivante.

Mon Projet Symfony2

Ce projet est propulsé par Symfony2, et construit grâce au tutoriel du siteduzero.

[Lire le tutoriel »](#)

Le blog
[Accueil du blog](#)
[Ajouter un article](#)

Blog

Liste des articles

- [Test](#) par dfg, le 01/01/2007
- [Lala](#) par winzou, le 01/01/2007

1	2	3
---	---	---

The sky's the limit © 2012 and beyond.

Nos articles et la pagination s'affichent

Bingo !

Améliorer les vues

Dans cette section, on s'est surtout intéressé au contrôleur. Mais on peut également améliorer nos vues, en les mutualisant entre certaines actions par exemple. L'idée que j'ai en tête, c'est d'avoir une unique vue `article.html.twig` qu'on pourra utiliser sur la page d'accueil ainsi que sur la page d'un article. Bien sûr ce n'est valable que si vous voulez la même présentation pour les deux, ce qui est mon cas.

Essayez donc de créer un `article.html.twig` générique, et de l'inclure depuis les deux autres vues.

`Article.html.twig`

Dans cette vue générique, on va intégrer l'affichage des informations de l'image et des catégories, dont on a fait la jointure au début de cette section. Voici ma version :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/article.html.twig #}

{# On utilise une variable temporaire, qu'on définit à false si
elle n'est pas déjà définie #}
{% set accueil = accueil|default(false) %}

<h2>
    {# On vérifie qu'une image soit bien associée à l'article #}
    {% if article.image is not null %}
        
    {% endif %}

    {# Si on est sur la page d'accueil, on fait un lien vers
    l'article, sinon non #}
    {% if accueil %}
        <a href="{{ path('sdzblog_voir', {'id': article.id}) }}>{{ article.titre }}</a>
    {% else %}
        {{ article.titre }}
    {% endif %}
</h2>

<i>Le {{ article.date|date('d/m/Y') }}, par {{ article.auteur }}.</i>

<div class="well">
    {{ article.contenu }}
</div>

{# On affiche les catégories éventuelles #}
{% if article.categories.count > 0 %}
    <div class="well well-small">
        <p><i>
            Catégories :
            {% for categorie in article.categories %}
                {{ categorie.nom }}{% if not loop.last %}, {% endif %}
            {% endfor %}
        </i></p>
    </div>
{% endif %}
```

Ce que vous pouvez faire pour améliorer cette vue :

- Utiliser le `slug` plutôt que l'`id` pour les liens vers les articles, histoire d'avoir un meilleur référencement ;
- Utiliser un [bundle de Markdown](#) pour mettre en forme le texte du contenu de l'article ;
- Utiliser les microdata pour améliorer la compréhension de votre page par les moteurs de recherche, notamment avec le format [Article](#).

N'hésitez pas à traiter ces points vous-même pour vous entraîner et pour améliorer le rendu de votre blog !

`Index.html.twig`

Dans cette vue, il faut juste modifier la liste afin d'utiliser la vue précédente pour afficher chaque article. Voici ma version :

Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig #}

{# ... La liste devient : #}
<ul>
    {% for article in articles %}
        {# On inclut la vue à chaque itération dans la boucle #}
```

```

    {%- include "SdzBlogBundle:Blog:article.html.twig" with
{'accueil': true} %}
    <hr />
    {%- else %}
        <p>Pas (encore !) d'articles</p>
    {%- endfor %}
</ul>

{# ... #}

```

Il faut toujours vérifier que la vue incluse aura les variables qu'elle attend. Ici, la vue `article.html.twig` utilise la variable `{ { article } }`, il faut donc :

- Que cette variable existe dans la vue qui l'inclut ;
 - Ou que la vue qui l'inclut précise cette variable dans le tableau `with` :
- ```
{ { include "..." with {'article': ... } }.
```

Nous sommes dans le premier cas, la variable `{ { article } }` est définie par le `for` sur la liste des articles. Par contre, la variable `accueil` est transmise *via* le `with`.

### Voir.html.twig

Dans cette vue, il faut juste remplacer l'affichage en dur de l'article par l'inclusion de notre nouvelle vue. Voici ma version :

#### Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig #-}

{%- extends "SdzBlogBundle::layout.html.twig" %}

{%- block title %}
 Lecture d'un article - {{ parent() }}
{%- endblock %}

{%- block sdzblog_body %}

{# Ici, on inclut la vue #}
{%- include "SdzBlogBundle:Blog:article.html.twig" %}

<p>

 <i class="icon-chevron-left"></i>
 Retour à la liste

 <i class="icon-edit"></i>
 Modifier l'article

 <i class="icon-trash"></i>
 Supprimer l'article

</p>
{%- endblock %}

```

## Pour conclure

Et voilà le premier TP du tutoriel s'achève ici. J'espère que vous avez pu exploiter toutes les connaissances que vous avez pu acquérir jusqu'ici, et qu'il vous a aidé à vous sentir plus à l'aise.

La prochaine partie du tutoriel va vous emmener plus loin avec Symfony2, pour connaitre tous les détails qui vous permettront de créer votre site internet de toute pièce. A bientôt !

## En résumé

- Vous savez maintenant construire vos entités ;
- Vous savez maintenant développer un contrôleur abouti ;
- Vous savez maintenant faire des jointures et plus encore dans vos repository ;
- Vous savez maintenant inclure habilement des vues pour éviter de dupliquer du code ;
- Bravo !

## Partie 4 : Allons plus loin avec Symfony2

### Créer des formulaires avec Symfony2

Quoi de plus important sur un site Web que les formulaires ? En effet, les formulaires sont l'interface entre vos visiteurs et votre contenu. Chaque commentaire, chaque article de blog, etc., tous passent par l'intermédiaire d'un visiteur et d'un formulaire pour exister dans votre base de données.

L'objectif de ce chapitre est donc de vous donner enfin les outils pour créer efficacement ces formulaires grâce à la puissance du composant `Form` de Symfony2. Ce chapitre va de pair avec le prochain, dans lequel nous parlerons de la validation des données, celles que vos visiteurs vont rentrer dans vos nouveaux formulaires.

#### Gestion des formulaires

#### L'enjeu des formulaires

Vous avez déjà créé des formulaires en HTML et PHP, vous savez donc que c'est une vraie galère ! À moins d'avoir créé vous-même un système dédié, gérer correctement des formulaires s'avère être un peu mission impossible. Par correctement, j'entends de façon maintenable, mais surtout, **réutilisable**. Heureusement, le composant `Form` de Symfony2 arrive à la rescousse !



N'oubliez pas que les composants peuvent être utilisés hors d'un projet Symfony2. Vous pouvez donc reprendre le composant `Form` dans votre site même si vous n'utilisez pas Symfony2.

### Un formulaire Symfony2, qu'est-ce que c'est ?

La vision Symfony2 sur les formulaires est la suivante : un formulaire se construit sur un objet existant, et son objectif est d'hydrater cet objet.

#### *Un objet existant*

Il nous faut donc des objets de base avant de créer des formulaires. Mais en fait, ça tombe bien : on les a déjà, ces objets ! En effet, un formulaire pour ajouter un article de blog va se baser sur l'objet `Article`, objet que nous avons construit lors du chapitre précédent. Tout est cohérent.



Je dis bien « objet » et non « entité Doctrine2 ». En effet, les formulaires n'ont pas du tout besoin d'une entité pour se construire, mais uniquement d'un simple objet. Heureusement, nos entités sont de simples objets avant d'être des entités, donc elles conviennent parfaitement. Je précise ce point pour vous montrer que les composants Symfony2 sont indépendants les uns des autres.

Pour la suite de ce chapitre, nous allons utiliser notre objet `Article`. C'est un exemple simple qui va nous permettre de construire notre premier formulaire. Je rappelle son code, sans les annotations pour plus de clarté (et parce qu'elles ne nous regardent pas ici) :

Code : PHP

```
<?php
namespace Sdz\BlogBundle\Entity;

use Sdz\BlogBundle\Entity\Tag;

class Article
{
 private $id;
 private $date;
 private $titre;
 private $contenu;
 private $auteur;
 private $publication;
 private $image;
 private $categories;
 private $commentaires;

 public function __construct()
 {
 $this->date = new \Datetime;
 $this->publication = true;
 $this->categories = new
\Doctrine\Common\Collections\ArrayCollection();
 $this->commentaires = new
\Doctrine\Common\Collections\ArrayCollection();
 }

 // ... Les getter et setter
}
```



Rappel : la convention pour le nom des *getters/setters* est importante : lorsque l'on parlera du champ « auteur », le composant Form utilisera l'objet *via* les méthodes `setAuteur()` et `getAuteur()` (comme le faisait Doctrine2 de son côté). Donc si vous aviez eu `set_auteur()` ou `recuperer_auteur()`, ça n'aurait pas fonctionné.

### Objectif : hydrater cet objet

Hydrater ? Un terme précis pour dire que le formulaire va remplir les attributs de l'objet avec les valeurs entrées par le visiteur. Faire `<?php $article->setAuteur('winzou') ?>`, `<?php $article->setDate(new \Datetime())`, etc., c'est *hydrater* l'objet Article.

Le formulaire en lui-même n'a donc comme seul objectif que d'hydrater un objet. Ce n'est qu'une fois l'objet hydraté que vous pourrez en faire ce que vous voudrez : enregistrer en base de données dans le cas de notre objet Article, envoyer un mail dans le cas d'un objet Contact, etc. Le système de formulaire ne s'occupe pas de ce que vous faites de votre objet, il ne fait que l'hydrater.

Une fois que vous avez compris ça, vous avez compris l'essentiel. Le reste n'est que de la syntaxe à connaître.

## Gestion basique d'un formulaire

Concrètement, pour créer un formulaire, il nous faut deux choses :

- un objet (on a toujours notre objet Article) ;
- un moyen pour construire un formulaire à partir de cet objet, un `FormBuilder`, « constructeur de formulaire » en français.

Pour faire des tests, placez-vous dans l'action `ajouterAction()` de notre contrôleur Blog et modifiez-la comme suit :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

use Sdz\BlogBundle\Entity\Article;

// ...

public function ajouterAction()
{
 // On crée un objet Article
 $article = new Article();

 // On crée le FormBuilder grâce à la méthode du contrôleur
 $formBuilder = $this->createFormBuilder($article);

 // On ajoute les champs de l'entité que l'on veut à notre
 // formulaire
 $formBuilder
 ->add('date', 'date')
 ->add('titre', 'text')
 ->add('contenu', 'textarea')
 ->add('auteur', 'text')
 ->add('publication', 'checkbox');
 // Pour l'instant, pas de commentaires, catégories, etc., on les
 // gérera plus tard

 // À partir du formBuilder, on génère le formulaire
 $form = $formBuilder->getForm();

 // On passe la méthode createView() du formulaire à la vue afin
 // qu'elle puisse afficher le formulaire toute seule
 return $this->render('SdzBlogBundle:Blog:ajouter.html.twig',
 array(
 'form' => $form->createView(),
));
}
```

Pour le moment, ce formulaire n'est pas opérationnel. On va pouvoir l'afficher, mais il ne se passera rien lorsqu'on va le valider.

Mais avant cette étape, essayons de comprendre le code présenté. Dans un premier temps, on récupère le `FormBuilder`. Cet objet n'est pas le formulaire en lui-même, c'est un constructeur de formulaire. On lui dit : « Crée un formulaire autour de l'objet `$article` », puis : « Ajoute les champs `date`, `titre`, `contenu` et `auteur` ». Et enfin : « Maintenant, donne-moi le formulaire construit avec tout ce que je t'ai dit avant ».

Prenons le temps de bien faire la différence entre les attributs de l'objet hydraté et les champs du formulaire. Un formulaire n'est pas du tout obligé d'hydrater tous les attributs d'un objet. On pourrait très bien ne pas utiliser le pseudo pour l'instant par exemple, et ne pas mettre de champ `auteur` dans notre formulaire. L'objet lui contient toujours l'attribut `auteur`, mais il ne sera juste pas hydraté par le formulaire. Bon en l'occurrence ce n'est pas le comportement que l'on veut (on va considérer le pseudo

comme obligatoire pour un article), mais sachez que c'est possible. 😊 D'ailleurs si vous avez l'œil, vous avez remarqué qu'on n'ajoute pas de champ `id` : comme il sera rempli automatiquement par Doctrine (grâce à l'auto-incrémation), le formulaire n'a pas besoin de remplir cet attribut.

Enfin, c'est avec cet objet `$form` généré que l'on pourra gérer notre formulaire : vérifier qu'il est valide, l'afficher, etc. Par exemple, ici, on utilise sa méthode `<?php $form->createView()` qui permet à la vue d'afficher ce formulaire. Concernant l'affichage du formulaire, j'ai une bonne nouvelle pour vous : Symfony2 nous permet d'afficher un formulaire simple en une seule ligne HTML ! Si si ; rendez-vous dans la vue `Blog/formulaire.html.twig` et ajoutez ces quelques lignes là où nous avions laissé un trou :

#### Code : HTML & Django

```
{# src/Sdz/BlogBundle/Resources/views/Blog/formulaire.html.twig #}

<h3>Formulaire d'article</h3>

<div class="well">
 <form method="post" {{ form_enctype(form) }}>
 {{ form_widget(form) }}
 <input type="submit" class="btn btn-primary" />
 </form>
</div>
```

Ensuite, admirez le résultat à l'adresse suivante : [http://localhost/Symfony/web/app\\_dev.php/blog/ajouter](http://localhost/Symfony/web/app_dev.php/blog/ajouter), impressionnant non ?

## Ajouter un article

### Formulaire d'article

Date

Titre

Contenu

Auteur

Publication

Valider

Aperçu du formulaire simple

Vous ajoutez un nouvel article.

Grâce à la fonction Twig `{{ form_widget() }}` on peut afficher un formulaire entier en une seule ligne. Alors bien sûr il n'est pas forcément à votre goût pour le moment, mais voyez le bon côté des choses : pour l'instant on est en plein développement, on veut tester notre formulaire. On s'occupera de l'esthétique plus tard. N'oubliez pas également de rajouter les balises `<form>` HTML et le bouton de soumission, car la fonction n'affiche que l'intérieur du formulaire.

Bon évidemment, comme je vous l'ai dit, ce code ne fait qu'afficher le formulaire, il n'est pas encore question de gérer sa soumission. Mais patience, on y arrive.



La date sélectionnée par défaut est celle d'aujourd'hui, et la checkbox Publication est déjà cochée : comment ça se fait ?

Bonne question ! Il est important de savoir que ces deux points ne sont pas là par magie, et que dans Symfony2 tout est cohérent. Si vous vous rappelez, on avait défini des valeurs dans le constructeur de l'entité Article :

#### Code : PHP

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

// ...

public function __construct()
{
 $this->date = new \Datetime;
 $this->publication = true;
 // ...
}
```

C'est à ce moment qu'est définie la valeur de ces deux attributs, et c'est sur la valeur de ces attributs que se base le formulaire pour remplir ses champs. Voilà l'origine de ces valeurs !

### Ajouter des champs

Vous pouvez le voir, ajouter des champs à un formulaire se fait assez facilement avec la méthode `<?php $formBuilder->add()` du `FormBuilder`. Les arguments sont les suivants :

1. Le nom du champ ;
2. Le type du champ ;
3. Les options du champ, sous forme de tableau.

Par « type de champ », il ne faut pas comprendre « type HTML » comme `text`, `password` ou `select`. Il faut comprendre « type sémantique ». Par exemple, le type `date` que l'on a utilisé affiche trois champs `select` à la suite pour choisir le jour, le mois et l'année. Il existe aussi un type `timezone` pour choisir le fuseau horaire. Bref, il en existe pas mal et ils n'ont rien à voir avec les types HTML, ils vont bien plus loin que ces derniers ! N'oubliez pas, Symfony2 est magique ! 

Voici l'ensemble des types de champ disponibles. Je vous dresse ici la liste avec pour chacun un lien vers la documentation : allez-y à chaque fois que vous avez besoin d'utiliser tel ou tel type.

Texte	Choix	Date et temps	Divers	Multiple	Caché
<ul style="list-style-type: none"> <li>• <code>text</code></li> <li>• <code>textarea</code></li> <li>• <code>email</code></li> <li>• <code>integer</code></li> <li>• <code>money</code></li> <li>• <code>number</code></li> <li>• <code>password</code></li> <li>• <code>percent</code></li> <li>• <code>search</code></li> <li>• <code>url</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>choice</code></li> <li>• <code>entity</code></li> <li>• <code>country</code></li> <li>• <code>language</code></li> <li>• <code>locale</code></li> <li>• <code>timezone</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>date</code></li> <li>• <code>datetime</code></li> <li>• <code>time</code></li> <li>• <code>birthday</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>checkbox</code></li> <li>• <code>file</code></li> <li>• <code>radio</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>collection</code></li> <li>• <code>repeated</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>hidden</code></li> <li>• <code>csrf</code></li> </ul>



Ayez bien cette liste en tête : le choix d'un type de champ adapté à l'attribut de l'objet sous-jacent est une étape importante dans la création d'un formulaire.

## Gestion de la soumission d'un formulaire

Afficher un formulaire c'est bien, mais faire quelque chose lorsqu'un visiteur le soumet, c'est quand même mieux !

- Pour gérer l'envoi du formulaire, il faut tout d'abord vérifier que la requête est de type POST : cela signifie que le visiteur est arrivé sur la page en cliquant sur le bouton `submit` du formulaire. Lorsque c'est le cas, on peut traiter notre formulaire.
- Ensuite, il faut faire le lien entre les variables de type POST et notre formulaire, pour que les variables de type POST viennent remplir les champs correspondants du formulaire. Cela se fait via la méthode `bind()` du formulaire. Cette méthode dit au formulaire : « Voici la requête d'entrée (nos variables de type POST entre autres). Lis cette requête, récupère les valeurs qui t'intéressent et hydrate l'objet ». Comme vous pouvez le voir, elle fait beaucoup de choses !
- Enfin, une fois que notre formulaire a lu ses valeurs et hydraté l'objet, il faut tester ces valeurs pour vérifier qu'elles sont valides avec ce que l'objet attend. Il faut valider notre objet. Cela se fait via la méthode `isValid()` du formulaire.

Ce n'est qu'après ces trois étapes que l'on peut traiter notre objet hydraté : sauvegarder en base de données, envoyer un email, etc.

Vous êtes un peu perdu ? C'est parce que vous manquez de code. Voici comment faire tout ce que l'on vient de dire, dans le contrôleur :

### Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

use Sdz\BlogBundle\Entity\Article;
```

```
// ...

public function ajouterAction()
{
 $article = new Article;

 // J'ai raccourci cette partie, car plus rapide à écrire !
 $form = $this->createFormBuilder($article)
 ->add('date', 'date')
 ->add('titre', 'text')
 ->add('contenu', 'textarea')
 ->add('auteur', 'text')
 ->add('publication', 'checkbox')
 ->getForm();

 // On récupère la requête
 $request = $this->get('request');

 // On vérifie qu'elle est de type POST
 if ($request->getMethod() == 'POST') {
 // On fait le lien Requête <-> Formulaire
 // A partir de maintenant, la variable $article contient les
 // valeurs entrées dans le formulaire par le visiteur
 $form->bind($request);

 // On vérifie que les valeurs rentrées sont correctes
 // (Nous verrons la validation des objets en détail dans le
 // prochain chapitre)
 if ($form->isValid()) {
 // On l'enregistre notre objet $article dans la base de
 // données
 $em = $this->getDoctrine()->getManager();
 $em->persist($article);
 $em->flush();

 // On redirige vers la page de visualisation de l'article
 // nouvellement créé
 return $this->redirect($this->generateUrl('sdzblog_voir',
 array('id' => $article->getId())));
 }
 }

 // À ce stade :
 // - soit la requête est de type GET, donc le visiteur vient
 // d'arriver sur la page et veut voir le formulaire
 // - soit la requête est de type POST, mais le formulaire n'est
 // pas valide, donc on l'affiche de nouveau

 return $this->render('SdzBlogBundle:Blog:ajouter.html.twig',
 array(
 'form' => $form->createView(),
));
}
```

Si le code paraît long, c'est parce que j'ai mis plein de commentaires ! Prenez le temps de bien le lire et de bien le comprendre : vous verrez, c'est vraiment simple. N'hésitez pas à le tester. Essayez de ne pas remplir un champ pour observer la réaction de Symfony2. Vous voyez que ce formulaire gère déjà très bien les erreurs, il n'enregistre l'article que lorsque tout va bien.



N'hésitez pas à tester votre formulaire en ajoutant des articles ! Il est opérationnel, et les articles que vous ajoutez sont réellement enregistrés en base de données.

Si vous l'avez bien testé, vous vous êtes rendu compte qu'on est obligé de cocher le champ `publication`. Ce n'est pas tellement le comportement voulu, car on veut pouvoir enregistrer un article sans forcément le publier (pour finir la rédaction plus tard par exemple). Pour cela, nous allons utiliser le troisième argument de la méthode `$formBuilder->add()` qui correspond aux options du champ. Les options se présentent sous la forme d'un simple tableau. Pour rendre le champ facultatif, il faut définir l'option `required` à `false`, comme suit :

#### Code : PHP

```
<?php
$formBuilder->add('publication', 'checkbox', array('required' =>
false));
```

Rappelez-vous donc : un champ de formulaire est requis par défaut. Si vous voulez le rendre facultatif, vous devez préciser l'option `required` à la main.

Un mot également sur la validation que vous rencontrez depuis le navigateur : impossible de valider le formulaire si un champ obligatoire n'est pas rempli :

Auteur  
winzou

Publication

**Veuillez cocher cette case si vous souhaitez continuer.**

Le champ Publication était obligatoire

Vous ajoutez un nouvel article.

Pourtant, nous n'avons pas utilisé de Javascript ! C'est juste du HTML5. En mettant l'attribut `required="required"` à une balise `<input>`, le navigateur interdit la validation du formulaire tant que cet input est vide. Pratique ! Mais attention, cela n'empêche pas de faire une validation côté serveur, au contraire. En effet, si quelqu'un utilise votre formulaire avec un vieux navigateur qui ne supporte pas le HTML5, il pourra valider le formulaire sans problème.

## Gérer les valeurs par défaut du formulaire

L'un des besoins courants dans les formulaires, c'est de mettre des valeurs prédéfinies dans les champs. Ça peut servir pour des valeurs par défaut (pré-remplir la date, par exemple) ou alors lors de l'édition d'un objet déjà existant (pour l'édition d'un article, on souhaite remplir le formulaire avec les valeurs de la base de données).

Heureusement, cela se fait très facilement. Il suffit de modifier l'instance de l'objet, ici `$article`, avant de le passer en argument à la méthode `createFormBuilder`, comme ceci :

Code : PHP

```
<?php
// On crée un nouvel article
$article = new Article;

// Ici, on pré-rempli avec la date d'aujourd'hui, par exemple
// Cette date sera donc pré-affichée dans le formulaire, cela
// facilite le travail de l'utilisateur
$article->setDate(new \Datetime());

// Et on construit le formBuilder avec cette instance d'article
$formBuilder = $this->createFormBuilder($article);

// N'oubliez pas d'ajouter les champs comme précédemment avec la
// méthode ->add()
```

Et si vous voulez modifier un article déjà enregistré en base de données, alors il suffit de le récupérer avant la création du formulaire, comme ceci :

Code : PHP

```
<?php
// Récupération d'un article déjà existant, d'id $id.
$article = $this->getDoctrine()
 ->getRepository('Sdz\BlogBundle\Entity\Article')
 ->find($id);

// Et on construit le formBuilder avec cette instance d'article,
// comme précédemment
$formBuilder = $this->createFormBuilder($article);

// N'oubliez pas d'ajouter les champs comme précédemment avec la
// méthode ->add()
```

## Personnaliser l'affichage d'un formulaire

Jusqu'ici, nous n'avons pas du tout personnalisé l'affichage de notre formulaire. Voyez quand même le bon côté des choses : on travaillait côté PHP, on a pu avancer très rapidement sans se soucier d'écrire les balises `<input>` à la main, ce qui est long et sans intérêt.

Mais bon, à un moment donné, il faut bien mettre la main à la pâte et faire des formulaires dans le même style que son site. Pour cela, je ne vais pas m'étendre, mais voici un exemple qui vous permettra de faire à peu près tout ce que vous voudrez :

Code : HTML

```

<form action="{{ path('votre_route') }}" method="post" {{ form_enctype(form) }}>

 <!-- Les erreurs générales du formulaire. -->
 {{ form_errors(form) }}

 <div>
 <!-- Génération du label. -->
 {{ form_label(form.titre, "Titre de l'article") }}

 <!-- Affichage des erreurs pour ce champ précis. -->
 {{ form_errors(form.titre) }}

 <!-- Génération de l'input. -->
 {{ form_widget(form.titre) }}
 </div>

 <!-- Idem pour un autre champ. -->
 <div>
 {{ form_label(form.contenu, "Contenu de l'article") }}
 {{ form_errors(form.contenu) }}
 {{ form_widget(form.contenu) }}
 </div>

 <!-- Génération des champs pas encore écrits.
 Dans cet exemple, ça serait « date », « auteur » et « publication »
 mais aussi le champ CSRF (géré automatiquement par Symfony !)
 et tous les champs cachés (type « hidden »). -->
 {{ form_rest(form) }}

</form>
</pre>

```

Pour plus d'information concernant l'habillage des formulaires, je vous invite à consulter la [documentation à ce sujet](#). Cela s'appelle en anglais le *form theming*.



Qu'est-ce que le CSRF ?

Le champ CSRF, pour *Cross Site Request Forgeries*, permet de vérifier que l'internaute qui valide le formulaire est bien celui qui l'a affiché. C'est un moyen de se protéger des envois de formulaire frauduleux. [Plus d'informations sur le CSRF](#). C'est un champ que Symfony2 rajoute automatiquement à tous vos formulaires, afin de les sécuriser sans même que vous vous en rendiez compte 😊.

## Créer des types de champs personnalisés

Il se peut que vous ayez envie d'utiliser un type de champ précis, mais que ce type de champ n'existe pas par défaut. Heureusement, vous n'êtes pas coincé, vous pouvez vous en sortir en créant votre propre type de champ. Vous pourrez ensuite utiliser ce champ comme n'importe quel autre dans vos formulaires.

Imaginons par exemple que vous n'aimiez pas le rendu du champ « date » avec ces trois balises `<select>` pour sélectionner le jour, le mois et l'année. Vous préféreriez un joli *datepicker* en JavaScript. La solution ? Créez un nouveau type de champ !

Je ne vais pas décrire la démarche ici, mais sachez que ça existe et que [la documentation traite ce point](#).

### Externaliser la définition de ses formulaires

Vous savez enfin créer un formulaire. Ce n'était pas très compliqué, nous l'avons rapidement fait et ce dernier se trouve être assez joli. Mais vous souvenez-vous de ce que j'avais promis au début : nous voulions un formulaire *réutilisable* ; or là, tout est dans le contrôleur, et je vois mal comment le réutiliser ! Pour cela, il faut détacher la définition du formulaire dans une classe à part, nommée `ArticleType` (par convention).

## Définition du formulaire dans ArticleType

`ArticleType` n'est pas notre formulaire. Comme tout à l'heure, c'est notre *constructeur de formulaire*. Par convention, on va mettre tous nos `xxxType.php` dans le répertoire `Form` du bundle. En fait, on va encore utiliser le générateur ici, qui sait générer les `FormType` pour nous, et vous verrez qu'on y gagne !

Exécutez donc la commande suivante :

#### Code : Console

```
php app/console doctrine:generate:form SdzBlogBundle:Article
```

Comme vous pouvez le voir c'est une commande Doctrine, car c'est lui qui a toutes les informations sur notre objet `Article`. Maintenant, vous pouvez aller voir le résultat dans le fichier `src/Sdz/BlogBundle/Form/ArticleType.php`.

On va commencer tout de suite par améliorer ce formulaire. En effet vous pouvez voir que les types de champ ne sont pas

précisés : le composant `Form` va les deviner à partir des annotations Doctrine qu'on a mis dans l'objet. Ce n'est pas une bonne pratique car cela peut être source d'erreur, c'est pourquoi je vous invite dès maintenant à remettre explicitement les types comme on avait déjà fait dans le contrôleur :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Form/ArticleType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ArticleType extends AbstractType
{
 public function buildForm(FormBuilderInterface $builder, array
$options)
 {
 $builder
->add('date', 'date')
->add('titre', 'text')
->add('contenu', 'textarea')
->add('auteur', 'text')
->add('publication', 'checkbox', array('required' => false))
 }

 public function setDefaultOptions(OptionsResolverInterface
$resolver)
 {
 $resolver->setDefaults(array(
 'data_class' => 'Sdz\BlogBundle\Entity\Article'
));
 }

 public function getName()
 {
 return 'sdz_blogbundle_articletype';
 }
}
```



J'ai également supprimé les champs `image` et `categories`, que nous verrons différemment plus loin dans ce chapitre.

Comme vous pouvez le voir, on n'a fait que déplacer la construction du formulaire, du contrôleur à une classe externe. Cet `ArticleType` correspond donc en fait à la définition des champs de notre formulaire. Ainsi, si l'on utilise le même formulaire sur plusieurs pages différentes, on utilisera ce même `ArticleType`. Fini le copier-coller ! Voilà la ré-utilisabilité. 😊

Rappelez-vous également, un formulaire se construit autour d'un objet. Ici, on a indiqué à Symfony2 quel était cet objet grâce à la méthode `setDefaultOptions()`, dans laquelle on a défini l'option `data_class`.

## Le contrôleur épuré

Avec cet `ArticleType`, la construction du formulaire côté contrôleur s'effectue grâce à la méthode `createForm()` du contrôleur (et non plus `createFormBuilder()`). Cette méthode utilise le composant `Form` pour construire un formulaire à partir du `ArticleType` passé en argument. Depuis le contrôleur, on récupère donc directement un formulaire, on ne passe plus par le constructeur de formulaire comme précédemment. Voyez par vous-même :

**Code : PHP**

```
<?php
// Dans le contrôleur

$article = new Article;
$form = $this->createForm(new ArticleType, $article);
```

En effet, si l'on s'est donné la peine de créer un objet à l'extérieur du contrôleur, c'est pour que ce contrôleur soit plus simple. C'est réussi !

Au final, en utilisant cette externalisation et en supprimant les commentaires, voilà à quoi ressemble la gestion d'un formulaire dans Symfony2 :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

use Sdz\BlogBundle\Entity\Article;
// N'oubliez pas de rajouter le ArticleType
use Sdz\BlogBundle\Form\ArticleType;

// ...

public function ajouterAction()
{
 $article = new Article;
 $form = $this->createForm(new ArticleType, $article);

 $request = $this->get('request');
 if($request->getMethod() == 'POST')
 {
 $form->bind($request);
 if($form->isValid())
 {
 $em = $this->getDoctrine()->getEntityManager();
 $em->persist($article);
 $em->flush();

 return $this->redirect($this->generateUrl('sdzblog_accueil'));
 }
 }

 return $this->render('SdzBlogBundle:Blog:ajouter.html.twig',
array(
 'form' => $form->createView(),
));
}
```

Plutôt simple, non ? Au final, votre code métier, votre code qui fait réellement quelque chose se trouve là où l'on a utilisé l'EntityManager. Pour l'exemple, nous n'avons fait qu'enregistrer l'article en base de données, mais c'est ici que vous pourrez envoyer un email, ou effectuer toute autre action dont votre site Internet aura besoin.

## Les formulaires imbriqués

### Intérêts de l'imbrication



Pourquoi imbriquer des formulaires ?

C'est souvent le cas lorsque vous avez des relations entre vos objets : vous souhaitez ajouter un objet A, mais en même temps un autre objet B qui est lié au premier. Exemple concret : vous voulez ajouter un client à votre application, votre Client est lié à une Adresse, mais vous avez envie d'ajouter l'adresse sur la même page que votre client, depuis le même formulaire. S'il fallait deux pages pour ajouter client puis adresse, votre site ne serait pas très ergonomique. Voici donc toute l'utilité de l'imbrication des formulaires !

## Un formulaire est un champ

Eh oui, voici tout ce que vous devez savoir pour imbriquer des formulaires entre eux. Considérez un de vos formulaires comme un champ, et appelez ce simple champ depuis un autre formulaire ! Bon, facile à dire, mais il faut savoir le faire derrière.

D'abord, créez le formulaire de notre entité Image. Vous l'aurez compris, on peut utiliser le générateur ici, exécutez donc cette commande :

**Code : Console**

```
php app/console doctrine:generate:form SdzBlogBundle:Image
```

En explicitant les types des champs, ça donne le code suivant :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Form/ImageType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ImageType extends AbstractType
{
```

```

public function buildForm(FormBuilderInterface $builder, array
$options)
{
 $builder
 ->add('url', 'text')
 ->add('alt', 'text')
 ;
}

public function setDefaultOptions(OptionsResolverInterface
$resolver)
{
 $resolver->setDefaults(array(
 'data_class' => 'Sdz\BlogBundle\Entity\Image'
));
}

public function getName()
{
 return 'sdz_blogbundle_imagetyp';
}
}

```

Ensuite, il existe deux façons d'imbriquer ce formulaire :

1. Avec une relation simple où l'on imbrique une seule fois un sous-formulaire dans le formulaire principal. C'est le cas le plus courant, celui de notre Article avec une seule Image ;
2. Avec une relation multiple, où l'on imbrique plusieurs fois le sous-formulaire dans le formulaire principal. C'est le cas d'un Client qui pourrait enregistrer plusieurs Adresse.

## Relation simple : imbriquer un seul formulaire

C'est le cas le plus courant, qui correspond à notre exemple de l'Article et de son Image. Pour imbriquer un seul formulaire en étant cohérent avec une entité, il faut que l'entité du formulaire principal (ici, Article) ait une relation *OneToOne* ou *ManyToOne* avec l'entité (ici, Image) dont on veut imbriquer le formulaire.

Une fois que vous savez ça, on peut imbriquer nos formulaires. C'est vraiment simple : allez dans ArticleType et rajoutez un champ image (du nom de la propriété de notre entité), de type... ImageType, bien sûr !

### Code : PHP

```

<?php
// src/Sdz/BlogBundle/Form/ArticleType.php

class ArticleType extends AbstractType
{
 public function buildForm(FormBuilderInterface $builder, array
$options)
 {
 $builder
 ->add('date', 'date')
 ->add('titre', 'text')
 ->add('contenu', 'textarea')
 ->add('auteur', 'text')
 ->add('publication', 'checkbox', array('required' => false))
 ->add('image', new ImageType()) // Rajoutez cette ligne
 ;
 }

 // ...
}

```

C'est tout ! Allez sur la page d'ajout : [/blog/ajouter](#). Le formulaire est déjà à jour (voir figure suivante), avec une partie « Image » où l'on peut remplir les deux seuls champs de ce formulaire, les champs « url » et « alt ». C'était d'une facilité déconcertante, n'est-ce pas ?

## Ajouter un article

### Formulaire d'article

Date

2012  10

04

Titre

Contenu

Auteur

Publication

Image

Url

Alt

**Valider**

Le formulaire est à jour

Réfléchissons bien sur ce qu'on vient de faire.

D'un côté nous avons l'entité Article qui possède un attribut image. Cet attribut image contient lui un objet Image. Il ne peut pas contenir autre chose, à cause du setter associé : celui-ci force l'argument à être un objet de classe Image.

L'objectif du formulaire est donc de venir injecter dans cet attribut image un objet Image, et pas autre chose ! On l'a vu au début de ce chapitre, un formulaire de type XxxType retourne un objet de classe Xxx. Il est donc tout à fait logique de mettre dans ArticleType, un champ image de type ImageType.

Sachez qu'il est bien entendu possible d'imbriquer les formulaires à l'infini de cette façon. La seule limitation c'est de faire quelque chose de compréhensible pour vos visiteurs, ce qui est tout de même le plus important.

Si lorsque vous validez votre formulaire vous avez une erreur de ce type :

#### Code : Console

```
A new entity was found through the relationship 'Sdz\BlogBundle\Entity\Article#cate
for entity: Sdz\BlogBundle\Entity\Categorie@000000000579b29e0000000061a76c55. To sc
on this unknown entity or configure cascade persist this association in the mapping
find out which entity causes the problem implement 'Sdz\BlogBundle\Entity\Categorie'
```

... c'est que Doctrine ne sait pas quoi faire avec l'entité Image qui est dans l'entité Article, car vous ne lui avez pas dit de persister cette entité. Pour résoudre l'erreur, il faut dire à Doctrine de persister cet objet Image bien sûr, suivez simplement les indications du message d'erreur :

- Soit vous ajoutez manuellement un \$em->persist (\$article->getImage ()) dans le contrôleur, avant le flush () ;
- Soit vous ajoutez une option à l'annotation @ORM\OneToOne dans l'entité Article, ce que nous avons fait si vous suivez ce cours depuis le début, comme ceci :

#### Code : Autre

```
/**
```

```
* @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image", cascade={"persist"})
*/
private $image;
```

C'est fini pour l'imbrication simple d'un formulaire dans un autre. Passons maintenant à l'imbrication multiple.

## Relation multiple : imbriquer un même formulaire plusieurs fois

On imbrique un même formulaire plusieurs fois lors que deux entités sont en relations *ManyToOne* ou *ManyToMany*.

On va prendre l'exemple ici de l'imbrication de plusieurs *CategorieType* dans le *ArticleType* principal. Attention cela veut dire qu'à chaque ajout d'*Article*, on aura la possibilité de créer de nouvelles *Categorie*. Ce n'est pas le comportement classique qui consiste plutôt à sélectionner des *Categorie* existantes. Ce n'est pas grave c'est pour l'exemple, sachant que plus loin dans ce chapitre on étudie également la manière de sélectionner ces catégories.

Tout d'abord, créez le formulaire *CategorieType* grâce au générateur :

### Code : Console

```
php app/console doctrine:generate:form SdzBlogBundle:Categorie
```

Voici ce que cela donne après avoir explicité les champs encore une fois :

### Code : PHP

```
<?php
// src/Sdz/BlogBundle/Form/CategorieType.php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class CategorieType extends AbstractType
{
 public function buildForm(FormBuilderInterface $builder, array
$options)
 {
 $builder
->add('nom', 'text')
 }

 public function setDefaultOptions(OptionsResolverInterface
$resolver)
 {
 $resolver->setDefaults(array(
 'data_class' => 'Sdz\BlogBundle\Entity\Categorie'
));
 }

 public function getName()
 {
 return 'sdz_blogbundle_categorietype';
 }
}
```

Maintenant, il faut rajouter le champ *categories* dans le *ArticleType*. Il faut pour cela utiliser le type *collection* et lui passer quelques options, comme ceci :

### Code : PHP

```
<?php
// src/Sdz/BlogBundle/Form/ArticleType.php

class ArticleType extends AbstractType
{
 public function buildForm(FormBuilderInterface $builder, array
$options)
 {
 $builder
 ->add('date', 'date')
 ->add('titre', 'text')
 ->add('contenu', 'textarea')
```

```

 ->add('auteur', 'text')
 ->add('publication', 'checkbox', array('required' => false))
 ->add('image', new ImageType())
 /*
 * Rappel :
 ** - 1er argument : nom du champ, ici « categories » car c'est le
nom de l'attribut
** - 2e argument : type du champ, ici « collection » qui est une
liste de quelque chose
** - 3e argument : tableau d'options du champ
*/
->add('categories', 'collection', array('type' => new
CategorieType(),
'allow_add' => true,
'allow_delete' => true,
'by_reference' => false))
;
}

// ...
}

```

On a ici utilisé le type de champ `collection`, qui permet en réalité de construire une collection (une liste) de n'importe quoi. Ici on a dit grâce à l'option `type` qu'il doit créer une liste de `Categorie`, mais on aurait pu faire une liste de type `text` : le formulaire aurait donc injecté dans l'attribut `categories` un simple tableau de textes.

Ce champ de type `collection` comporte plusieurs options en plus du type. Vous notez les options `allow_add` et `allow_delete`, qui autorisent au formulaire d'ajouter des entrées en plus dans la collection, ainsi que d'en supprimer. En effet, on pourrait tout à fait ne pas autoriser ces actions, ce qui aurait pour effet de ne pouvoir modifier que les `Categorie` qui sont déjà liées à l'`Article`.

Assez parlé, testons dès maintenant le résultat. Pour cela, actualisez la page d'ajout d'un article. Ah mince, le mot « `Categorie` » est bien inscrit, mais il n'y a rien en dessous. 😞 Ce n'est pas un bug, c'est bien voulu par Symfony2. En effet, comme l'entité `Article` lié au formulaire de base n'a pas encore de catégories, le champ `collection` n'a rien à afficher ! Et si on veut créer des catégories, il ne peut pas savoir à l'avance combien on veut en créer : 1, 2, 3 ou plus ?

La solution, sachant qu'on doit pouvoir ajouter à l'infini, et même supprimer, est d'utiliser du JavaScript. OK, ça ne nous fait pas peur !

D'abord, affichez la source de la page et regardez l'étrange balise `<div>` que Symfony2 a rajoutée en dessous du label `Categorie`:

#### Code : HTML

```

<div id="sdz_blogbundle_articletype_categories" data-
prototype="<div&gt<label
class="required">_name_label__</label>
<div
id="sdz_blogbundle_articletype_categories__name_"><div&gt
<label for="sdz_blogbundle_articletype_categories__name__nom">
class="required">Nom</label><input
type="text">
id="sdz_blogbundle_articletype_categories__name__nom"
name="sdz_blogbundle_articletype[categories][__name__][nom]"
required="true" />
</div></div></div>">
</div>

```

Notez surtout l'attribut `data-prototype`. C'est en fait un attribut (au nom arbitraire) rajouté par Symfony2 et qui contient ce à quoi doit ressembler le code HTML pour ajouter un tag. Essayez de le lire, vous voyez qu'il contient les balises `<label>` et `<input>`, tout ce qu'il faut pour créer le champ `nom` compris dans `CategorieType`, en fait.

Du coup, on le remercie, car grâce à ce template, ajouter des champs en JavaScript est un jeu d'enfant. Je vous propose un script JavaScript rapidement fait qui emploie la librairie jQuery et qui est à mettre dans notre fichier `Blog/formulaire.html.twig`:

#### Code : HTML & Django

```

{# src/Sdz/BlogBundle/Resources/views/Blog/formulaire.html.twig #-}

<form method="post" {{ form_enctype(form) }}>
{{ form_widget(form) }}

{# Ajout d'un lien pour ajouter un champ catégorie supplémentaire.
#}
Ajouter une
catégorie

<input type="submit" class="btn btn-primary" />


```

```

</form>
</div>

<!-- On charge la librairie jQuery. Ici, je la prends depuis le
site jquery.com, mais si vous l'avez en local, changez simplement
l'adresse. -->
<script src="http://code.jquery.com/jquery-1.8.2.min.js"></script>

<script type="text/javascript">
$(document).ready(function() {
 // On récupère la balise <div> en question qui contient
 l'attribut « data-prototype » qui nous intéresse
 var $container = $('#sdz_blogbundle_articletype_categories');

 // On définit une fonction qui va ajouter un champ
 function add_category() {
 // On définit le numéro du champ (en comptant le nombre de
 champs déjà ajoutés)
 index = $container.children().length;

 // On ajoute à la fin de la balise <div> le contenu de
 l'attribut « data-prototype »
 // Après avoir remplacé la variable __name__ qu'il contient par
 le numéro du champ
 $container.append($('#data-prototype').replace(/__name__/g, index));

 // On ajoute également un bouton pour pouvoir supprimer la
 catégorie
 $container.append($('<a href="#" id="delete_category_' + index +
 '" class="btn btn-danger">Supprimer

'));

 // On supprime le champ à chaque clic sur le lien de
 suppression
 $('#delete_category_' + index).click(function() {
 $(this).prev().remove(); // $(this).prev() est le template
 ajouté
 $(this).remove(); // $(this) est le lien de
 suppression
 return false;
 });
 }

 // On ajoute un premier champ directement s'il n'en existe pas
 déjà un (cas d'un nouvel article par exemple)
 if($container.children().length == 0) {
 add_category();
 }

 // On ajoute un nouveau champ à chaque clic sur le lien d'ajout
 $('#add_category').click(function() {
 add_category();
 return false;
 });
});
</script>

```

Appuyez sur F5 sur la page d'ajout et admirez le résultat (voir figure suivante). Voilà qui est mieux !

Le formulaire est opérationnel Le formulaire est opérationnel

Et voilà, votre formulaire est maintenant opérationnel ! Vous pouvez vous amuser à créer des articles contenant plein de nouvelles catégories en même temps.

Voilà ! Ce chapitre important n'était pas si compliqué dans le fond !

Bien entendu, vous ne pouvez pas vous arrêter en si bon chemin. Maintenant que vos formulaires sont opérationnels, il faut bien vérifier un peu ce que vos visiteurs vont y mettre comme données ! C'est l'objectif du prochain chapitre, qui traite de la validation des données justement. Il vient compléter le chapitre actuel, continuez donc la lecture !

## Validez vos données

Au chapitre précédent nous avons vu comment créer des formulaires avec Symfony2. Mais qui dit formulaire dit vérification des données rentrées ! Symfony2 contient un composant Validator qui, comme son nom l'indique, s'occupe de gérer tout cela. Attaquons-le donc !

### Pourquoi valider des données ?

#### Never Trust User Input

Ce chapitre introduit la validation des objets avec le composant *Validator* de Symfony2. En effet, c'est normalement un des premiers réflexes à avoir lorsque l'on demande à l'utilisateur de remplir des informations : vérifier ce qu'il a rempli ! Il faut toujours considérer que soit il ne sait pas remplir un formulaire, soit c'est un petit malin qui essaie de trouver la faille. Bref, ne jamais faire confiance à ce que l'utilisateur vous donne (« *never trust user input* » en anglais).

La validation et les formulaires sont bien sûr liés, dans le sens où les formulaires ont besoin de la validation. Mais l'inverse n'est pas vrai ! Dans Symfony2, le *validator* est un service indépendant et n'a nul besoin d'un formulaire pour exister. Ayez-le en tête, avec le *validator*, on peut valider n'importe quel objet, entité ou non, le tout sans avoir besoin de formulaire.

### L'intérêt de la validation

L'objectif de ce chapitre est donc d'apprendre à définir qu'une entité est valide ou non. Plus concrètement, il nous faudra établir des règles précises pour dire que tel attribut (le nom d'utilisateur par exemple) doit faire 3 caractères minimum, que tel autre attribut (l'âge par exemple) doit être compris entre 7 et 77 ans, etc. En vérifiant les données avant de les enregistrer en base de données, on est certain d'avoir une base de données cohérente, dans laquelle on peut avoir confiance !

### La théorie de la validation

La théorie, très simple, est la suivante. On définit des règles de validation que l'on va rattacher à une classe. Puis on fait appel à un service extérieur pour venir lire un objet (instance de ladite classe) et ses règles, et définir si oui ou non l'objet en question respecte ces règles. Simple et logique !

#### Définir les règles de validation

#### Les différentes formes de règles

Pour définir ces règles de validation, ou contraintes, il existe deux moyens :

1. Le premier est d'utiliser les annotations, vous les connaissez maintenant. Leur avantage est d'être situé au sein même de l'entité, et juste à côté des annotations du *mapping Doctrine2* si vous les utilisez également pour votre *mapping*.
2. Le deuxième est d'utiliser le Yaml, XML ou PHP. Vous placez donc vos règles de validation hors de l'entité, dans un fichier séparé.

Les deux moyens sont parfaitement équivalents en termes de fonctionnalités. Le choix se fait donc selon vos préférences. Dans la suite du cours, j'utiliserais les annotations car je trouve extrêmement pratique de centraliser règles de validation et *mapping Doctrine* au même endroit. Facile à lire et à modifier. 😊

### Définir les règles de validation

#### Préparation

Nous allons prendre l'exemple de notre entité `Article` pour construire nos règles. La première étape consiste à déterminer les règles que nous voulons avec des mots, comme ceci :

- La date doit être une date valide ;
- Le titre doit faire au moins 10 caractères de long ;
- Le contenu ne doit pas être vide ;
- L'auteur doit faire au moins 2 caractères de long ;
- L'image liée doit être valide selon les règles attachées à l'objet `Image` ;

À partir de cela, nous pourrons convertir ces mots en annotations.

#### Annotations

Pour définir les règles de validation, nous allons donc utiliser les annotations. La première chose à savoir est le namespace des annotations à utiliser. Souvenez-vous, pour le *mapping Doctrine* c'était `@ORM`, ici nous allons utiliser `@Assert`, donc le namespace complet est le suivant :

#### Code : PHP

```
<?php
use Symfony\Component\Validator\Constraints as Assert;
```

Ce `use` est à rajouter au début de l'objet que l'on va valider, notre entité `Article` en l'occurrence. En réalité, vous pouvez définir l'alias à autre chose qu'`Assert`. Mais c'est une convention qui s'est installée, donc autant la suivre pour avoir un code plus facilement lisible pour les autres développeurs.

Ensuite, il ne reste plus qu'à ajouter les annotations pour traduire les règles que l'on vient de lister. Sans plus attendre, voici donc la syntaxe à respecter. Exemple avec notre objet Article :

Code : PHP

```
<?php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
// N'oubliez pas de rajouter ce « use », il définit le namespace pour
les annotations de validation
use Symfony\Component\Validator\Constraints as Assert;

use Doctrine\Common\Collections\ArrayCollection;

/**
 * Sdz\BlogBundle\Entity\Article
 *
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 */
class Article
{
 /**
 * @var integer $id
 *
 * @ORM\Column(name="id", type="integer")
 * @ORM\Id
 * @ORM\GeneratedValue(strategy="AUTO")
 */
 private $id;

 /**
 * @var date $date
 *
 * @ORM\Column(name="date", type="date")
 * @Assert\DateTime()
 */
 private $date;

 /**
 * @var string $titre
 *
 * @ORM\Column(name="titre", type="string", length=255)
 * @Assert\MinLength(10)
 */
 private $titre;

 /**
 * @var text $contenu
 *
 * @ORM\Column(name="contenu", type="text")
 * @Assert\NotBlank()
 */
 private $contenu;

 /**
 * @var string $auteur
 *
 * @ORM\Column(name="auteur", type="string", length=255)
 * @Assert\MinLength(2)
 */
 private $auteur;

 /**
 * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image",
 * cascade={"persist"})
 * @Assert\Valid()
 */
 private $image;

 // ...
}
```

Vraiment pratique d'avoir les métadonnées Doctrine et les règles de validation au même endroit, n'est-ce pas ?



J'ai pris l'exemple ici d'une entité car ça sera souvent le cas. Mais n'oubliez pas que vous pouvez mettre des règles de validation sur n'importe quel objet, qui n'est pas forcément entité.

### Syntaxe

Revenons un peu sur les annotations que l'on a ajouté. Nous avons utilisé la forme simple, qui est construite comme ceci :

#### Code : Autre

```
@Assert\Contrainte(valeur de l'option par défaut)
```

Avec :

- La Contrainte, qui peut être, comme vous l'avez vu, *NotBlank* ou *MinLength*, etc. Nous voyons plus loin toutes les contraintes possibles.
- La Valeur entre parenthèses, qui est la valeur de l'option par défaut. En effet chaque contrainte à plusieurs options, dont une par défaut souvent intuitive. Par exemple, l'option par défaut de *MinLength* (longueur minimale en français) est évidemment la valeur de la longueur minimale que l'on veut appliquer, 10 pour le titre.

Mais on peut aussi utiliser la forme étendu qui permet de personnaliser la valeur de plusieurs options en même temps, comme ceci :

#### Code : Autre

```
@Assert\Contrainte(option1="valeur1", option2="valeur2", ...)
```

Les différentes options diffèrent d'une contrainte à une autre, mais voici un exemple avec la contrainte *MinLength* :

#### Code : Autre

```
@Assert\MinLength(limit=10, message="Le titre doit faire au moins {{ limit }} car
```



Oui, vous pouvez utiliser {{ limit }}, qui est en fait le nom de l'option que vous voulez faire apparaître dans le message.

Bien entendu, vous pouvez mettre plusieurs contraintes sur un même attribut. Par exemple pour un attribut numérique telle une note, on pourrait mettre les deux contraintes suivantes :

#### Code : PHP

```
<?php
/**
 * @Assert\Min(0)
 * @Assert\Max(20)
 */
private $note
```

Vous savez tout ! Il n'y a rien de plus à connaître sur les annotations. A part les contraintes existantes et leurs options évidemment.

#### Liste des contraintes existantes

Voici un tableau qui regroupe la plupart des contraintes, à avoir sous la main lorsque vous définissez vos règles de validation ! Elles sont bien entendu toutes documentées, donc n'hésitez pas à vous référer à la [documentation officielle](#) pour toute information supplémentaire.

Toutes les contraintes disposent de l'option "message", qui est le message à afficher lorsque la contrainte est violée. Je n'ai pas répété cette option dans les tableaux suivants, mais sachez qu'elle existe bien à chaque fois.

Contraintes de base :

Contrainte	Rôle	Options
NotBlank Blank	La contrainte NotBlank vérifie que la valeur soumise n'est ni une chaîne de caractères vide, ni NULL. La contrainte Blank fait l'inverse.	-
True False	La contrainte True vérifie que la valeur vaut true, 1 ou "1". La contrainte False vérifie que la valeur vaut false, 0 ou "0".	-

Type	La contrainte Type vérifie que la valeur est bien du type donné en argument.	<i>type</i> (Option par défaut) : Le type duquel doit être la valeur, parmi <code>array</code> , <code>bool</code> , <code>int</code> , <code>object</code> , etc.
------	------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------

Contraintes sur des chaînes de caractères :

Contrainte	Rôle	Options
Email	La contrainte Email vérifie que la valeur est une adresse email valide.	<i>checkMX</i> (Défaut : false) : Si défini à true, Symfony2 va vérifier les MX de l'email via la fonction <code>checkdnsr</code> .
MinLength MaxLength	La contrainte MinLength vérifie que la valeur donnée fait au moins X caractères de long. MaxLength, au plus X caractères de long.	<i>limit</i> (Option par défaut) : Le seuil de longueur, en nombre de caractères. <i>charset</i> (Défaut : UTF-8) : Le charset à utiliser pour calculer la longueur.
Url	La contrainte Url vérifie que la valeur est une adresse URL valide.	<i>protocols</i> (Défaut : <code>array('http', 'https')</code> ) : Définit les protocoles considérés comme valide. Si vous voulez accepter les URL en <code>ftp://</code> , ajoutez-le à cette option.
Regex	La contrainte Regex vérifie la valeur par rapport à une regex.	<i>pattern</i> (Option par défaut) : La regex à faire correspondre. <i>match</i> (Défaut : true) : Définit si la valeur doit (true) ou ne doit pas (false) correspondre à la regex.
Ip	La contrainte Ip vérifie que la valeur est une adresse IP valide.	<i>type</i> (Défaut : 4) : Version de l'IP à considérer. 4 pour IPv4, 6 pour IPv6, all pour toutes les versions, et d'autres.
Language	La contrainte Language vérifie que la valeur est un code de langage valide selon la norme.	-
Locale	La contrainte Locale vérifie que la valeur est une locale valide. Exemple : fr ou fr_FR.	-
Country	La contrainte Country vérifie que la valeur est un code pays en 2 lettres valide. Exemple : fr.	-

Contraintes sur les nombres :

Contrainte	Rôle	Options
Max Min	La contrainte Max vérifie que la valeur ne dépasse pas X. Min, que la valeur dépasse ce X.	<i>limit</i> (Option par défaut) : La valeur seuil. <i>invalidMessage</i> : Message d'erreur lorsque la valeur n'est pas un nombre.

Contraintes sur les dates :

Contrainte	Rôle	Options
Date	La contrainte Date vérifie que la valeur est un objet de type Datetime, ou une chaîne de caractères du type 'YYYY-MM-DD'.	-
Time	La contrainte Time vérifie que la valeur est un objet de type Datetime, ou une chaîne de caractères du type 'HH:MM:SS'.	-
Datetime	La contrainte Datetime vérifie que la valeur est un objet de type Datetime, ou une chaîne de caractères du type 'YYYY-MM-DD HH:MM:SS'.	-



Les noms de contraintes sont sensibles à la casse. Cela signifie que la contrainte `DateTime` existe, mais que `Datetime` ou `datetime` n'existent pas ! Soyez attentif à ce petit détail pour éviter des erreurs inattendues. 😊

## Créer ses propres contraintes

Sachez que vous pouvez créer vos propres contraintes en suivant [la documentation à ce sujet](#). Nous verrons également plus loin comment réaliser des "contraintes dynamiques" en utilisant une méthode de l'objet validé, très pratique !

### Déclencher la validation

#### Le service Validator

Comme je l'ai dit plus haut, ce n'est pas l'objet qui se valide tout seul, on doit déclencher la validation nous-même. Ainsi, vous pouvez tout à fait assigner une valeur non valide à un attribut sans qu'aucune erreur ne se déclenche. Par exemple, vous pouvez faire `<?php $article->setTitre('abc')` alors que ce titre a moins de 10 caractères, il est invalide.

Pour valider l'objet, on passe par un acteur externe : le service `validator`. Ce service s'obtient comme n'importe quel autre service :

Code : PHP

```
<?php
// Depuis un contrôleur

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

Ensuite, on doit demander à ce service de valider notre objet. Cela se fait grâce à la méthode `validate` du service. Cette méthode retourne un tableau qui est soit vide si l'objet est valide, soit rempli des différentes erreurs lorsque l'objet n'est pas valide. Pour bien comprendre, exécutez cette méthode dans un contrôleur :

#### Code : PHP

```
<?php
// Depuis un contrôleur

// ...

public function testAction()
{
 $article = new Article;

 $article->setDate(new \Datetime()); // Champ « date » O.K.
 $article->setTitre('abc'); // Champ « titre »
incorrect : moins de 10 caractères.
 // $article->setContenu('blabla'); // Champ « contenu »
incorrect : on ne le définit pas.
 $article->setAuteur('A'); // Champ « auteur »
incorrect : moins de 2 caractères.

// On récupère le service validator.
$validator = $this->get('validator');

// On déclenche la validation.
$liste_erreurs = $validator->validate($article);

// Si le tableau n'est pas vide, on affiche les erreurs.
if(count($liste_erreurs) > 0) {
 return new Response(print_r($liste_erreurs, true));
} else {
 return new Response("L'article est valide !");
}
}
```

Vous pouvez vous amuser avec le contenu de l'entité `Article` pour voir comment réagit le validateur 😊.

## La validation automatique sur les formulaires

Vous devez savoir qu'en pratique, on ne se servira que très peu du service `validator` nous-mêmes. En effet, le formulaire de Symfony2 le fait à notre place ! Nous venons de voir le fonctionnement du service `validator` pour comprendre comment l'ensemble marche, mais en réalité on l'utilisera très peu de cette manière.

Rappelez-vous le code pour la soumission d'un formulaire :

#### Code : PHP

```
<?php
// Depuis un contrôleur

if ($request->getMethod() == 'POST') {
 $form->bind($request);

 if ($form->isValid()) {
 // ...
 }
}
```

Avec la ligne 7, le formulaire `$form` va lui-même faire appel au service `validator`, et valider l'objet qui vient d'être hydraté par le formulaire. Derrière cette ligne se cache donc le code que nous avons vu au paragraphe précédent. Les erreurs sont assignées au formulaire, et sont affichées dans la vue. Nous n'avons rien à faire, pratique !

## Conclusion

Cette sous-partie a pour objectif de vous faire comprendre ce qu'il se passe déjà lorsque vous utilisez la méthode `isValid` d'un formulaire. De plus, vous savez qu'il est possible de valider un objet indépendamment de tout formulaire, en mettant la main à la pâte.

## Encore plus de règles de validation

### Valider depuis un Getter

Vous le savez, un getter est une méthode qui commence le plus souvent par "get", mais qui peut également commencer par "is". Le composant Validation accepte les contraintes sur les attributs, mais également sur les getters ! C'est très pratique, car vous pouvez alors mettre une contrainte sur une fonction, avec toute la liberté que cela vous apporte.

Tout de suite un exemple d'utilisation :

**Code : PHP**

```
<?php
class Article
{
 // ...
 /**
 * @Assert\True()
 */
 public function isArticleValid()
 {
 return false;
 }
}
```

Cet exemple vraiment basique considère l'article comme non valide, car l'annotation `@Assert\True()` attend que la méthode retourne `true` alors qu'elle retourne `false`. Vous pouvez l'essayer dans votre formulaire, vous verrez le message "Cette valeur doit être vraie" (message par défaut de l'annotation `True()`) s'affiche en haut du formulaire. C'est donc une erreur qui s'applique à l'ensemble du formulaire.

Mais il existe un moyen de déclencher une erreur liée à un champ en particulier, ainsi l'erreur s'affichera juste à côté de ce champ. Il suffit de nommer le getter "`is + le nom d'un attribut`" : par exemple "`isTitre`" si l'on veut valider le titre. Essayez par vous-même le code suivante :

**Code : PHP**

```
<?php
class Article
{
 // ...
 /**
 * @Assert\True()
 */
 public function isTitre()
 {
 return false;
 }
}
```

Vous verrez que l'erreur "Cette valeur doit être vraie" s'affiche bien à côté du champ titre.

Bien entendu, vous pouvez faire plein de traitements et de vérifications dans cette méthode, ici j'ai juste mis "`return false`" pour l'exemple 😊. Je vous laisse imaginer les possibilités.

## Valider intelligemment un attribut objet

Derrière ce titre se cache une problématique toute simple : lorsque je valide un objet A, comment valider un objet B en attribut, d'après ses propres règles de validation ?

Il faut utiliser la contrainte `Valid`, qui va déclencher la validation du sous objet B selon les règles de validation de cet objet B. Prenons un exemple :

**Code : PHP**

```
<?php
class A
{
 /**
 * @Assert\MinLength(5)
 */
 private $titre;

 /**
 * @Assert\Valid()
 */
 private $b;
}

class B
{
 /**
 * @Assert\Min(10)
 */
}
```

```
 private $nombre;
}
```

Avec cette règle, lorsqu'on déclenche la validation sur l'objet A, le service `validator` va valider l'attribut `titre` selon le `MinLength()`, puis va aller chercher les règles de l'objet B pour valider l'attribut `nombre` de B selon le `Min()`. N'oubliez pas cette contrainte, car valider un sous-objet n'est pas le comportement par défaut : sans cette règle dans notre exemple, vous auriez pu sans problème ajouter une instance de B qui ne respecte pas la contrainte de 10 minimum pour son attribut `nombre`. Vous pourriez donc rencontrer des petits soucis de logique si vous l'oubliez 😊.

## Valider depuis un Callback

L'objectif de la contrainte `Callback` est d'être personnalisable à souhait. En effet, des fois vous pouvez avoir besoin de valider des données selon votre propre logique, qui ne rentre pas dans un `Maxlength`.

L'exemple classique est la censure de mots non désirés dans un attribut texte. Reprenons notre Article, et considérons que l'attribut `contenu` ne peut pas contenir les mots "échec" et "abandon". Voici comment mettre en place une règle qui va rendre invalide le contenu s'il contient l'un de ces mots :

**Code : PHP**

```
<?php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
use Doctrine\Common\Collections\ArrayCollection;
use Sdz\BlogBundle\Entity\Tag;

// On rajoute ce use pour le context :
use Symfony\Component\Validator\ExecutionContext;

/**
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 *
 * @Assert\Callback(methods={"contenuValide"})
 */
class Article
{
 // ...

 public function contenuValide(ExecutionContext $context)
 {
 $mots_interdits = array('échec', 'abandon');

 // On vérifie que le contenu ne contient pas l'un des mots
 if (preg_match('#'.implode('|', $mots_interdits). '#', $this->getContenu())) {
 // La règle est violée, on définit l'erreur
 et son message
 // 1er argument : on dit quel attribut l'erreur concerne, ici
 "contenu"
 // 2e argument : le message d'erreur
 $context->addViolationAtSubPath('contenu', 'Contenu invalide car il
 contient un mot interdit.', array(), null);
 }
 }
}
```

Vous auriez même pu aller plus loin en comparant des attributs entre eux, par exemple pour interdire le pseudo dans un mot de passe. L'avantage du Callback par rapport à une simple contrainte sur un getter, c'est de pouvoir ajouter plusieurs erreurs à la fois, en définissant sur quel attribut chacun se trouve grâce au premier argument de la méthode `addViolationAtSubPath` (en mettant "contenu" ou "titre", etc). Souvent la contrainte sur un getter suffira, mais pensez à ce Callback pour les fois où vous serez limité 😊.

## Valider un champ unique

Il existe une dernière contrainte très pratique : `UniqueEntity`. Cette contrainte permet de valider que la valeur d'un attribut est unique parmi toutes les entités existantes. Pratique pour vérifier qu'une adresse email n'existe pas déjà dans la base de données par exemple.

Vous avez bien lu, j'ai parlé d'`entité`. En effet c'est une contrainte un peu particulière car elle ne se trouve pas dans le composant Validator, mais dans le Bridge entre Doctrine et Symfony2 (ce qui fait le lien entre ces deux bibliothèques). On n'utilisera donc pas `@Assert\UniqueEntity` mais simplement `@UniqueEntity`. Il faut bien sûr en contrepartie faire attention de rajouter ce `use` à chaque fois que vous l'utilisez :

**Code : Autre**

```
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

Voici comment on pourrait, dans notre exemple avec Article, contraindre nos titres à être tous différents les uns des autres :

#### Code : PHP

```
<?php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
use Doctrine\Common\Collections\ArrayCollection;
use Sdz\BlogBundle\Entity\Tag;

// On rajoute ce use pour la contrainte :
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Table()
 *
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\ArticleRepository")
 *
 * @UniqueEntity(fields="titre", message="Un article existe déjà avec
 * ce titre.")
 */
class Article
{
 // ... Les autres contraintes ne changent pas, pas même celle(s)
 sur l'attribut titre

 // Mais pour être logique, il faudrait aussi mettre la colonne
 titre en Unique pour Doctrine :
 /**
 * @ORM\Column(name="titre", type="string", length=255, unique=true)
 */
 private $titre;
}
```

## Valider selon nos propres contraintes

Vous commencer à vous habituer : avec Symfony2 il est possible de tout faire ! L'objectif de ce paragraphe est d'apprendre à créer notre propre contrainte, que l'on pourra utiliser en annotation : `@NotreContrainte`. L'avantage d'avoir sa propre contrainte est double :

- D'une part c'est une contrainte réutilisable sur vos différents objets : on pourra l'utiliser sur Article mais également sur Commentaire, etc. ;
- D'autre part cela permet de placer le code de validation dans un objet externe... et surtout dans un service ! Indispensable, vous comprendrez.

Une contrainte est toujours lié à un validateur, qui va être en mesure de valider la contrainte. Nous allons donc les faire en deux étapes. Pour l'exemple, nous allons créer une contrainte AntiFlood, qui impose un délai de 15 secondes entre chaque message posté sur le site.

### 1. Créer la contrainte

Tout d'abord, il faut créer la contrainte en elle-même : c'est celle que nous appellerons en annotation depuis nos objets. Une classe de contrainte est vraiment très basique, tout la logique se trouvera dans le validateur en réalité.

Je vous invite donc simplement à créer le fichier suivant :

#### Code : PHP

```
<?php
// src/Sdz/BlogBundle/Validator/AntiFlood.php

namespace Sdz\BlogBundle\Validator;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class AntiFlood extends Constraint
{
 public $message = 'Vous avez déjà posté un message il y a moins
 de 15 secondes, merci d\'attendre un peu.';
}
```



L'annotation `@Annotation` est nécessaire pour que cette nouvelle contrainte soit disponible via les annotations dans les autres classes. En effet, toutes les classes ne sont pas des annotations, heureusement 😊

Les options de l'annotation correspondent aux attributs publics de la classe d'annotation. Ici, on a l'attribut "message", on pourra donc faire

**Code : Autre**

```
@AntiFlood(message="Mon message personnalisé")
```

C'est tout pour la contrainte ! Passons au validateur.

## 2. Créer le validateur

C'est la contrainte qui décide par quel validateur elle doit se faire valider. Par défaut, une contrainte `xxx` demande à se faire valider par le validateur `XxxValidator`. Créons donc le validateur `AntiFloodValidator` :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Validator/AntiFloodValidator.php

namespace Sdz\BlogBundle\Validator;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class AntiFloodValidator extends ConstraintValidator
{
 public function validate($value, Constraint $constraint)
 {
 // Pour l'instant, on considère comme flood tout message de
 moins de 3 caractères
 if (strlen($value) < 3) {
 // C'est cette ligne qui déclenche l'erreur pour le
 formulaire, avec en argument le message
 $this->context->addViolation($constraint->message);
 }
 }
}
```

C'est tout pour le validateur. Il n'est pas très compliqué non plus, il contient juste une méthode `validate()` qui permet de valider ou non la valeur. Son argument `$value` correspond à la valeur de l'attribut sur laquelle on a défini l'annotation. Par exemple si l'on avait défini l'annotation comme ceci :

**Code : Autre**

```
/**
 * @AntiFlood()
 */
private $contenu;
```

Alors c'est tout logiquement le contenu de l'attribut "`$contenu`" au moment de la validation qui sera injecté en tant qu'argument "`$value`".

La méthode `validate()` ne doit pas renvoyer `true` ou `false` pour confirmer que la valeur est valide ou non. Elle doit juste lever une `Violation` si la valeur est invalide. C'est ce qu'on a fait ici dans le cas où la chaîne fait moins de 3 caractères : on ajoute une violation, dont l'argument est le message d'erreur.

Sachez aussi que vous pouvez utiliser des messages d'erreur avec des paramètres. Par exemple : "Votre message %string% est considéré comme flood". Pour définir ce paramètre `%string%` utilisé dans le message, il faut le passer dans le 2<sup>e</sup> argument de la méthode `addViolation`, comme ceci :

**Code : PHP**



```
<?php
$this->context->addViolation($constraint->message,
array('%string%' => $value));
```

Et voilà vous savez créer votre propre contrainte ! Pour l'utiliser, c'est comme n'importe quelle autre annotation : on importe le namespace de l'annotation, et on la met en commentaire juste avant l'attribut concerné, voici un exemple sur l'entité `Commentaire` :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Entity/Commentaire.php
```

```

namespace Sdz\BlogBundle\Entity\Commentaire;

use Sdz\BlogBundle\Validator\AntiFlood;

class Commentaire
{
 // ...

 /**
 * @ORM\Column(name="contenu", type="text")
 * @AntiFlood()
 */
 private $contenu;

 // ...
}

```

Votre annotation sera ainsi pris en compte au même titre qu'un `@Assert\MaxLength(x)` par exemple !

Mais si vous avez bien suivi, vous savez qu'on n'a pas encore vu le principal intérêt de nos propres contraintes : la validation par un service !

### 3. Transformer son validateur en service



Qu'est-ce qu'un service ?

Un service, on l'a déjà rapidement mentionné, c'est un objet qui remplit une fonction et qui peut être accédé de presque n'importe où dans votre code Symfony2. Je ne vais pas décrire ici tout leur fonctionnement, car [un chapitre leur est dédié](#) plus loin dans ce tutoriel. On va juste voir très brièvement comment s'en servir dans le cadre d'une contrainte.



Quel est l'intérêt d'utiliser un service pour valider une contrainte ?

L'intérêt, vous ne le savez pas encore, c'est qu'un service peut accéder à toute sorte d'information utile. Il suffit de créer un service, de lui "injecter" les données, il pourra ainsi s'en servir. Dans notre cas, on va lui injecter la requête et l'*EntityManager* comme données : il pourra ainsi valider notre contrainte non seulement à partir de la valeur `$value` d'entrée, mais également en fonction de paramètres extérieurs qu'on ira chercher dans la base de données !

#### 3.1. Définition du service

Prenons un exemple pour bien comprendre le champ des possibilités. Tout d'abord, rentrez la définition du service dans le fichier `services.yml` dans votre bundle :

**Code : YAML**

```

src/Sdz/BlogBundle/Resources/config/services.yml

services:
 sdzblob.validator.antiflood: # Le
 nom du service
 class: Sdz\BlogBundle\Validator\AntiFloodValidator # La
 classe du service, ici notre validateur déjà créé
 arguments: [@request, @doctrine.orm.entity_manager] # Les
 données qu'on injecte au service : la requête et l'EntityManager
 scope: request # Comme
 on injecte la requête, on doit préciser ce scope
 tags:
 - { name: validator.constraint_validator, alias:
 sdzblob_antiflood } # C'est avec l'alias qu'on retrouvera le
 service

```



Ne faites pas attention à toute cette syntaxe de création de service. C'est normal si vous ne la comprenez pas : je ne vous l'ai pas encore introduite ! Copiez la simplement, ce qui nous intéresse dans ce chapitre est ailleurs 😊.



Si le fichier `services.yml` n'existe pas déjà chez vous, c'est qu'il n'est pas chargé automatiquement. Pour cela, il faut faire une petite manipulation je vous invite à lire le début du [chapitre sur les services](#).

#### 3.2. Modification de la contrainte

Maintenant que notre validateur est un service en plus d'être simplement un objet, nous devons adapter un petit peu notre code. Tout d'abord, modifions la contrainte pour qu'elle demande à se faire valider par le service d'alias "sdzblob\_antiflood" et non plus simplement par l'objet classique `AntiFloodValidator`. Pour cela, il suffit de lui rajouter la méthode `validateBy()` suivante :

**Code : PHP**

```
<?php
```

```
// src/Sdz/BlogBundle/Validator/AntiFlood.php

namespace Sdz\BlogBundle\Validator;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class AntiFlood extends Constraint
{
 public $message = 'Vous avez déjà posté un message il y a moins
de 15 secondes, merci d\'attendre un peu.';

 public function validatedBy()
 {
 return 'sdzblog_antiflood'; // Ici, on fait appel à l'alias du
 service
 }
}
```

### 3.3. Modification du validateur

Enfin, il faut adapter notre validateur pour que d'une part il récupère les données qu'on lui injecte, grâce au constructeur, et d'autre part qu'il s'en serve tout simplement :

Code : PHP

```
<?php

// src/Sdz/BlogBundle/Validator/AntiFloodValidator.php

namespace Sdz\BlogBundle\Validator;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

use Doctrine\ORM\EntityManager;
use Symfony\Component\HttpFoundation\Request;

class AntiFloodValidator extends ConstraintValidator
{
 private $request;
 private $em;

 // Les arguments déclarés dans la définition du service
 // arrivent au constructeur
 // On doit les enregistrer dans l'objet pour pouvoir s'en
 // resservir dans la méthode validate()
 public function __construct(Request $request, EntityManager $em)
 {
 $this->request = $request;
 $this->em = $em;
 }

 public function validate($value, Constraint $constraint)
 {
 // On récupère l'IP de celui qui poste
 $ip = $this->request->server->get('REMOTE_ADDR');

 // On vérifie si cette IP a déjà posté un message il y a
 // moins de 15 secondes
 $isFlood = $this->em-
 >getRepository('SdzBlogBundle:Commentaire')
 ->isFlood($ip, 15); // Bien entendu, il
 // faudrait écrire cette méthode isFlood, c'est pour l'exemple

 if (strlen($value) < 3 && $isFlood) {
 // C'est cette ligne qui déclenche l'erreur pour le
 // formulaire, avec en argument le message
 $this->context->addViolation($constraint->message);
 }
 }
}
```

Et voilà nous venons de faire une contrainte qui s'utilise aussi facilement qu'une annotation, et qui pourtant fait un gros travail en allant chercher dans la base de données si l'IP courante envoie trop de messages. Un peu de travail à la création de la contrainte, mais son utilisation est un jeu d'enfant à présent !



Vous trouverez un brin plus d'information sur la page de la documentation sur la création de contrainte, notamment comment faire une contrainte qui s'applique non pas à un attribut, mais à une classe entière.

## Pour conclure

Vous savez maintenant valider dignement vos données, félicitations !

Le formulaire était le dernier point que vous aviez vraiment besoin d'apprendre. À partir de maintenant, vous pouvez créer un site Internet en entier avec Symfony2, il ne manque plus que la sécurité à aborder, car pour l'instant, sur notre blog, tout le monde peut tout faire. 😊

Rendez-vous au prochain chapitre pour régler ce petit détail. 😊

## En résumé

- Le composant Validation permet de valider les données d'un objet suivant des règles définies ;
- Cette validation est systématique lors de la soumission d'un formulaire : il est en effet impensable de laisser l'utilisateur rentrer ce qu'il veut sans vérifier !
- Les règles de validation se définissent *via* les annotations directement à côté des attributs à valider de la classe. Vous pouvez bien sûr utiliser d'autres formats tel que le yaml ou le xml ;
- Il est également possible de valider à l'aide de getter, de callback ou même de service. Cela rend la procédure de validation très flexible et très puissante.

## Sécurité et gestion des utilisateurs

Dans ce chapitre, nous allons apprendre la sécurité avec Symfony2. C'est un chapitre assez technique, mais indispensable : à la fin nous aurons un espace membre fonctionnel et sécurisé !

Nous allons avancer en deux étapes : la première sera consacrée à la théorie de la sécurité sous Symfony2. Nécessaire, elle nous permettra d'aborder la deuxième étape : l'installation du bundle FOSUserBundle, qui viendra compléter notre espace membre.

Bonne lecture !

### Authentification et autorisation

La sécurité sous Symfony2 est très poussée, vous pouvez la contrôler très finement, mais surtout très facilement. Pour atteindre ce but, Symfony2 a bien séparé deux mécanismes différents : l'authentification et l'autorisation. Prenez le temps de bien comprendre ces deux notions pour bien attaquer la suite du cours. 😊

### Les notions d'authentification et d'autorisation

#### L'authentification

L'authentification est le processus qui va définir qui vous êtes, en tant que visiteur. L'enjeu est vraiment très simple : soit vous ne vous êtes pas identifié sur le site et vous êtes un anonyme, soit vous vous êtes identifié (via le formulaire d'identification ou via un cookie "Se souvenir de moi") et vous êtes un membre du site. C'est ce que la procédure d'authentification va déterminer. Ce qui gère l'authentification dans Symfony2 s'appelle un *firewall*.

Ainsi vous pourrez sécuriser des parties de votre site internet juste en forçant le visiteur à être un membre authentifié. Si le visiteur l'est, le *firewall* va le laisser passer, sinon, le *firewall* va le rediriger sur la page d'identification. Cela se fera donc dans les paramètres du *firewall*, nous les verrons plus en détails par la suite.

#### L'autorisation

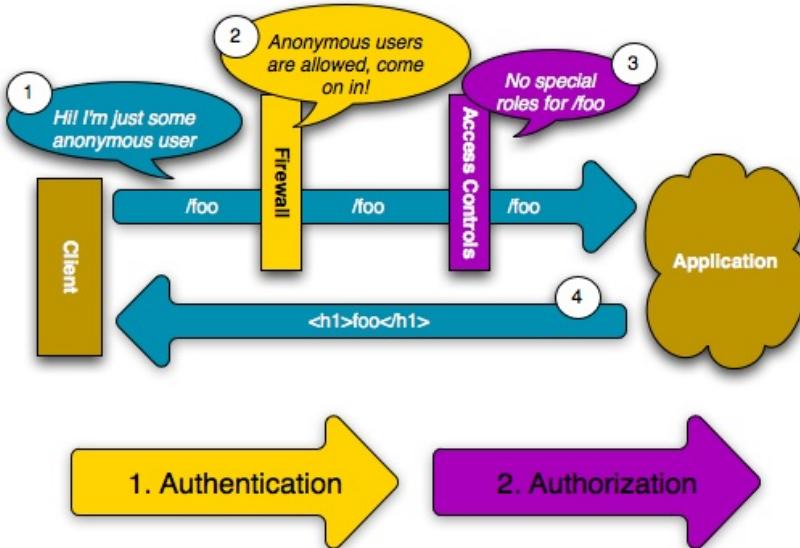
L'autorisation est le processus qui va déterminer si vous avez le droit de faire l'action que vous voulez faire. Il agit donc après le *firewall*. Ce qui gère l'autorisation dans Symfony2 s'appelle l'*Access Control*.

Par exemple, un membre identifié lambda aura accès à la liste de sujets d'un forum, mais ne pourra pas supprimer de sujet. Seuls les membres disposant des droits administrateurs le peuvent, ce que l'*Access Control* va vérifier.

### Exemples

Pour bien comprendre la différence, je reprends ici l'exemple de la [documentation officielle](#), qui est je trouve très bien illustré. Dans ces exemples, vous distinguerez bien les différents acteurs de la sécurité.

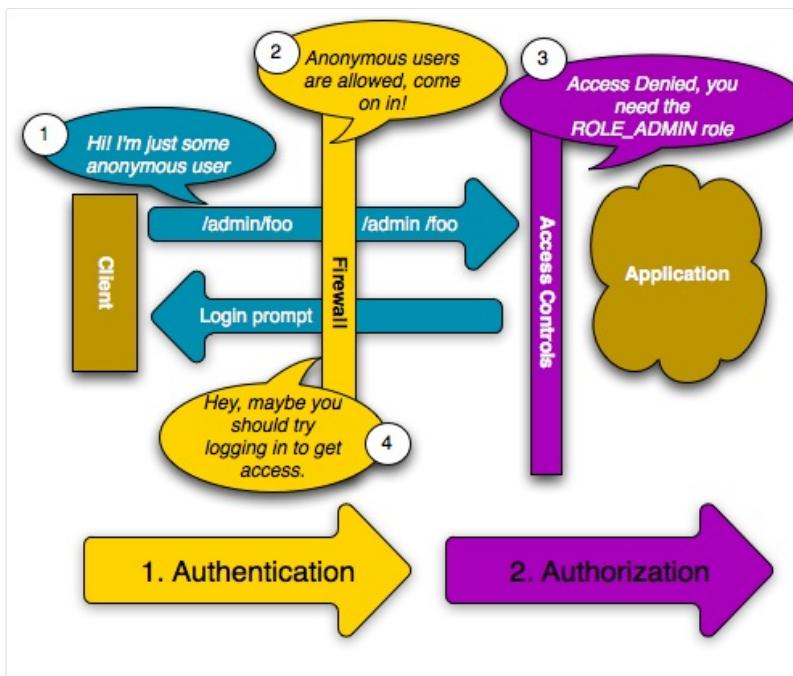
#### Accès à une page anonymement et sans droits particuliers



Sur ce schéma, vous distinguez bien le *firewall* d'un côté, et l'*Access Control* de l'autre. C'est très clair, mais reprenons le ensemble pour bien comprendre :

1. Le visiteur n'est pas identifié, et tente d'accéder à la page /foo ;
2. Le Firewall est configuré d'une manière qu'il n'est pas nécessaire d'être identifié pour accéder à la page /foo. Il laisse donc passer notre visiteur ;
3. L'Access Control regarde s'il y a besoin de droit particulier pour accéder à la page /foo : non il n'y en a pas. Il laisse donc passer notre visiteur ;
4. Le visiteur a atteint notre application, et il reçoit la page /foo en retour.

### Accès à une page anonymement mais qui nécessite des droits

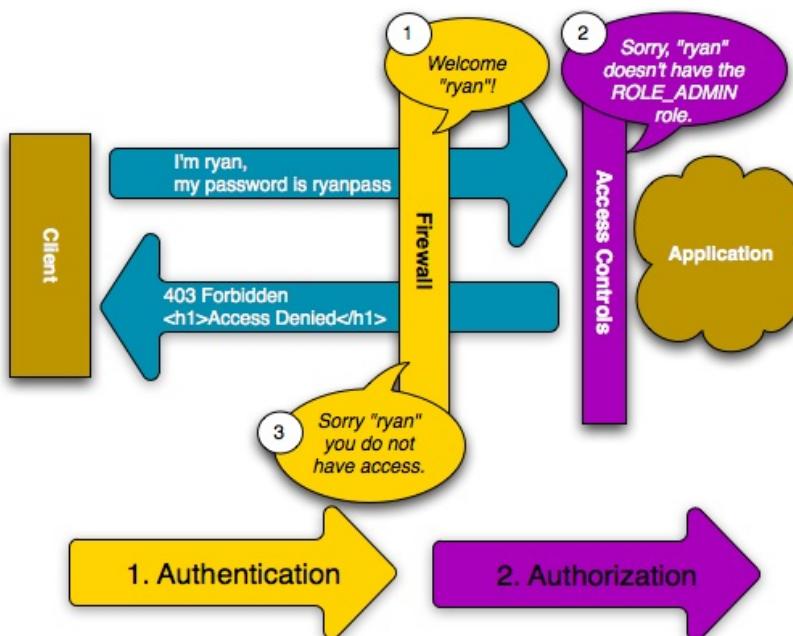


Dans ce cas précis, la page `/admin/foo` pourrait être visité de façon anonyme (le firewall ne bloque pas l'utilisateur), mais il faut par contre posséder les droits admin, représenté par le rôle `ROLE_ADMIN`. Usuellement on le verra, on bloque l'administration directement via le firewall, mais c'est un exemple. Voici le process pas à pas :

1. Le visiteur n'est pas identifié, et tente d'accéder à la page `/admin/foo` ;
2. Le Firewall est configuré d'une manière qu'il n'est pas nécessaire d'être identifié pour accéder à la page `/admin/foo`. Il laisse donc passer notre visiteur ;
3. L'Access Control regarde s'il y a besoin de droit particulier pour accéder à la page `/admin/foo` : oui, il faut le rôle `ROLE_ADMIN`. Le visiteur n'a pas ce rôle, donc l'Access Control lui bloque l'accès à la page `/admin/foo` ;
4. Le visiteur n'a donc pas atteint la page qu'il cherchait, et se fait renvoyer sur la page d'identification.

### Accès à une page de façon identifié mais sans les droits suffisants

Cet exemple est le même que précédemment, sauf que notre visiteur est identifié, il s'appelle Ryan.



1. Ryan s'identifie (via un cookie par exemple), et il tente d'accéder à la page `/admin/foo` ;  
D'abord, le firewall confirme l'authentification de Ryan (c'est son rôle !). Visuellement c'est bon. Ensuite, on ne sait pas ici de quelle manière est configuré le firewall pour la page `/admin/foo`, mais comme Ryan est identifié, il ne le bloque pas ;
2. L'Access Control regarde s'il y a besoin de droit particulier pour accéder à la page `/admin/foo` : oui, il faut le rôle `ROLE_ADMIN`, que Ryan n'a pas. Il bloque donc l'accès à la page `/admin/foo` à Ryan ;

3. Ryan n'a pas atteint la page qu'il cherchait, et se fait renvoyer sur une page d'erreur lui disant qu'il n'a pas les droits suffisants ("Accès interdit").

## Les différents acteurs de la sécurité Symfony2

### Le fichier de configuration de la sécurité

Maintenant qu'on a vu le processus global, il nous faut étudier plus en détail comment fonctionne chaque acteur. Pour cela, ouvrez le fichier `app/config/security.yml`, car tous les acteurs sont paramétrés dans ce fichier de configuration. Si vous ne l'avez pas encore modifié, voici à quoi il ressemble :

**Code : Autre**

```
app/config/security.yml

security:
 encoders:
 Symfony\Component\Security\Core\User\User: plaintext

 role_hierarchy:
 ROLE_ADMIN: ROLE_USER
 ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]

 providers:
 in_memory:
 users:
 user: { password: userpass, roles: ['ROLE_USER'] }
 admin: { password: adminpass, roles: ['ROLE_ADMIN'] }

 firewalls:
 dev:
 pattern: ^/(_profiler|wdt)|css|images|js/
 security: false

 login:
 pattern: ^/demo/secured/login$
 security: false

 secured_area:
 pattern: ^/demo/secured/
 form_login:
 check_path: /demo/secured/login_check
 login_path: /demo/secured/login
 logout:
 path: /demo/secured/logout
 target: /demo
 #anonymous: ~
 #http_basic:
 # realm: "Secured Demo Area"

 access_control:
 #-
 { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel: https }
 #-
 { path: ^/_internal, roles: IS_AUTHENTICATED_ANONYMOUSLY, ip: 127.0.0.1 }
```



Maintenant, voyons un à un les acteurs et leur configuration.

## Les acteurs de l'authentification

### Les firewalls

Comme on l'a vu, un `firewall` cherche à vérifier que vous êtes bien celui que vous prétendez être.

Je dis bien **un** `firewall` car on peut en définir plusieurs, selon l'URL sur votre site. Vous pouvez en distinguer trois dans le fichier de configuration : "dev", "login" et "secured\_area" (les noms sont arbitraires). Intérêt ? Exemple : j'ai un `firewall` "main" pour tout mon site, et un `firewall` "admin" pour la partie administration de mon site, dont l'URL commence par `/admin`. Cette multiplicité des `firewalls` est importante, car elle vous permettra de définir plusieurs zones de sécurité à votre site.

Enfin, chaque `firewall` a besoin d'être lié à une source d'utilisateurs. Ils ont besoin de savoir où aller chercher les utilisateurs, pour vérifier qu'ils sont bien identifiés. Ces sources d'utilisateurs peuvent être multiples : inscrits directement dans le fichier de configuration, depuis une entité Doctrine2, ou même depuis n'importe quelle autre source. Ces sources s'appellent des `providers`, nous en reparlons plus loin.

Comme vous pouvez un peu le deviner depuis le fichier de configuration, un `firewall` se définit par :

- Un **pattern**, qui est une expression régulière sur les URL à protéger avec ce `firewall`.
- Des informations de connexion, définies par **form\_login** dans le fichier de configuration. En effet un `firewall` a besoin de

quelques informations pour la connexion de nos membres :

- l'URL de la page de connexion (**login\_path**), c'est l'URL sur laquelle se trouve le formulaire de connexion pour ce firewall. Cette page n'est pas gérée par Symfony2, il faut soit la faire à la main soit utiliser un bundle qui va la faire pour nous. Nous utiliserons le bundle FOSUserBundle.
- l'URL cible du formulaire de connexion (**check\_path**), c'est l'URL à laquelle sera envoyé le formulaire de connexion, et qui vérifiera les identifiants renseignés par l'utilisateur. Cette page, elle, est déjà gérée par le *firewall*, nous n'aurons pas besoin de l'implémenter. Par contre, pour que Symfony2 sache où sont nos utilisateurs, il faut le lui dire via l'option *provider*.
- Un **provider**, c'est l'origine de nos utilisateurs. Ici cette option n'est pas renseigné, Symfony2 va donc prendre le seul *provider* défini dans la section *providers* un peu plus haut dans le fichier. On reparle de ces *providers* juste après, mais sachez qu'au même titre que les *firewalls*, on pourra avoir plusieurs *providers* (un par *firewall* par exemple).
- Et d'autres options que nous verrons plus loin.

Ne vous inquiétez pas de la syntaxe précise de chacune de ces options, on fait une synthèse du fichier security.yml plus loin.

### **Les providers**

Comme on l'a déjà vu, un *provider* est une source d'utilisateurs, là où les *firewalls* peuvent aller chercher les utilisateurs pour les identifier.

Pour l'instant vous pouvez le voir dans le fichier, un seul *provider* est défini, nommé "in\_memory" (encore une fois, le nom est arbitraire). C'est un *provider* assez particulier dans le sens où les utilisateurs sont directement listés dans ce fichier de configuration, il s'agit des utilisateurs "user" et "admin". Vous l'aurez compris, c'est un *provider* de développement, pour tester *firewalls* et autres sans avoir besoin d'une quelconque base de données derrière.

Heureusement il existe d'autres formes de *provider* :

- La première est de donner une entité à Symfony2, et le firewall saura aller chercher en base de données les utilisateurs en utilisant son *repository*. Nous n'utiliserons pas cette méthode mais sachez juste qu'elle existe, et [elle est documentée](#).
- La deuxième est d'utiliser ce qu'on appelle un *UserProvider*. C'est un service qui permet de personnaliser comment les *firewalls* accèdent à vos utilisateurs. Vous pouvez écrire ce service vous-même, mais sachez qu'il en existe un dans le bundle FOSUserBundle et qu'on n'aura pas à le faire. Concrètement, il ne contient en gros qu'une seule méthode qui retourne un utilisateur en fonction d'un nom d'utilisateur (via une requête en base de données par exemple). Un utilisateur est un objet quelconque qui implémente l'interface [Symfony\Component\Security\Core\User\UserInterface](#). Pour en savoir plus sur les UserProviders, je vous invite à lire le [cookbook de la documentation officielle](#), il est vraiment intéressant !

### **L'encoder**

*L'encoder* est un service qui va encoder les mots de passe des utilisateurs. Il permet de personnaliser la façon dont vous voulez hasher les mots de passe.

Vous pouvez deviner dans le fichier que *l'encoder* utilisé, "plaintext", n'encode en réalité rien du tout. Il stocke les mots de passe en clair, c'est pourquoi les mots de passe du *provider* "in\_memory" sont en clair 😊. Par la suite nous définirons un *encoder* du type sha512, une méthode sûre !

## **Les acteurs de l'autorisation**

### **Les rôles**

Les rôles sont des acteurs passifs de l'autorisation. La notions de rôle et autorisation est très simple : lorsqu'on va limiter l'accès à certains pages, on va se baser sur les rôles de l'utilisateur. Ainsi, limiter l'accès au panel d'administration revient à limiter cet accès aux utilisateurs disposant du rôle ROLE\_ADMIN (par exemple).

Le nom des rôles n'a pas d'importance, si ce n'est qu'il doit commencer par "ROLE\_".

Commençons par définir les rôles de l'application. Prenez un papier et listez tous les rôles dont vous aurez besoin. Par exemple pour notre blog, on aurait :

- ROLE\_ADMIN, pour les administrateurs du blog
- ROLE\_AUTEUR, pour les auteurs, ceux qui peuvent ajouter des articles au blog
- ROLE\_MODERATEUR, pour ceux qui peuvent modérer les commentaires

Vous pouvez en avoir d'autres évidemment !

Maintenant l'idée est de créer une hiérarchie entre ces rôles. Ainsi, on peut sans difficulté dire que le ROLE\_ADMIN comprend les ROLE\_AUTEUR et ROLE\_MODERATEUR. Pour limiter l'accès à certaines pages dans notre code, on ne va donc pas faire "Si l'utilisateur a ROLE\_AUTEUR ou s'il a ROLE\_ADMIN, alors il peut écrire un article". Mais grâce à la définition de la hiérarchie, on peut faire simplement "Si l'utilisateur a ROLE\_AUTEUR". Car un utilisateur qui dispose du ROLE\_ADMIN dispose également du ROLE\_AUTEUR, c'est une inclusion.

Ce sont ces relations, et uniquement ces relations, que nous allons inscrire dans le fichier app/config/security.yml.

### **L'Access Control**

Comme on l'a vu, l'Access Control va s'occuper de savoir si le visiteur a les bons droits pour accéder à la page demandée.

Il y a différents moyens d'utiliser l'access control :

- Soit depuis le fichier de configuration security.yml, en appliquant des règles sur des URL. Vous pouvez le voir dans le

fichier, les deux dernières lignes commentées servent à protéger des URL. On peut bien sûr utiliser des expressions régulières, ce qui permet d'appliquer une règle à un ensemble d'URL.

- Soit directement depuis les contrôleurs, en appliquant des règles sur des méthodes. On peut ainsi appliquer des règles différentes selon des paramètres, c'est libre. Il existe d'ailleurs le bundle [JMSSecurityExtraBundle](#), inclus par défaut dans Symfony2, qui permet de définir ces règles via le système d'annotations.

La différence entre ces deux moyens d'utiliser la même protection est importante, et offre en fait une flexibilité intéressante.

### Installation du bundle FOSUserBundle

Comme vous avez pu le voir, la sécurité fait intervenir de nombreux acteurs et demande un peu de travail. C'est normal, c'est un point sensible d'un site internet. Heureusement, d'autres développeurs talentueux ont réussi à nous faciliter la tâche en créant un bundle qui gère une partie de la sécurité !

Ce bundle s'appelle FOSUserBundle, il est très utilisé par la communauté Symfony2 car vraiment bien fait, et surtout répondant à un besoin vraiment basique d'un site internet : l'authentification des membres. Mettre en place ce bundle est par contre vraiment facile, rien à voir avec une méthode maison 😊.

Je vous propose donc d'installer ce bundle dans la suite de cette sous-partie.

### Télécharger FOSUserBundle

Le bundle FOSUserBundle est hébergé sur [www.github.com](https://github.com), comme beaucoup de bundles et projets Symfony2. Sa page est ici : <https://github.com/FriendsOfSymfony/FOSUserBundle>.

Mais pour ajouter ce bundle, vous l'avez compris, il faut utiliser Composer ! Commencez par déclarer cette nouvelle dépendance dans votre fichier `composer.json` :

**Code : JavaScript**

```
// composer.json
{
 // ...
 "require": {
 // ...
 "friendsofsymfony/user-bundle": "*"
 }
 // ...
}
```

Ensuite, il faut mettre à jour vos dépendances pour que Composer aille chercher ce nouveau bundle :

**Code : Console**

```
php composer.phar update friendsofsymfony/user-bundle
```



L'argument après la commande "update" permet de dire à Composer de ne mettre à jour que cette dépendance. Ici, cela permet de ne mettre à jour que FOSUserBundle, et pas les autres dépendances. C'est plus rapide, mais si vous voulez tout mettre à jour, supprimez simplement ce paramètre 😊.

### Activer le bundle

Si vos souvenirs sont bons, vous devriez savoir qu'un bundle ne s'active pas tout seul, il faut aller l'enregistrer dans le noyau de Symfony2. Pour cela, ouvrez le fichier `app/AppKernel.php` pour enregistrer le bundle :

**Code : PHP**

```
<?php
// app/AppKernel.php

public function registerBundles()
{
 $bundles = array(
 // ...
 new FOS\UserBundle\FOSUserBundle(),
);
}
```

C'est bon, le bundle est bien enregistré. Mais inutile d'essayer d'accéder à votre application Symfony2 maintenant, elle ne marchera pas. Il faut en effet faire un peu de configuration et de personnalisation avant de pouvoir tout remettre en marche.

### Personnaliser FOSUserBundle

Ce bundle est un bundle générique évidemment, car il doit pouvoir s'adapter à tout type d'utilisateur de n'importe quel site

internet. Vous imaginez bien que cela est impossible "out-of-the-box" ! Il faut donc s'atteler à faire un peu de personnalisation afin de faire correspondre le bundle à nos besoins. Cette personnalisation passe par la création d'un bundle qui va hériter de FOSUserBundle.

### Créer notre propre SdzUserBundle

On va pour cela créer notre propre bundle, qui va hériter de FOSUserBundle, et on va ainsi pouvoir personnaliser à peu près tout. Créer un petit bundle est vraiment simple, et vous devez savoir le faire à ce stade. Attention ici on n'utilise pas le générateur, car il génère trop de fichiers par rapport à ce que l'on veut ici. On va donc le faire totalement à la main, voici les étapes :

- Créez le répertoire `src/Sdz/UserBundle` ;
- Mettez un fichier `SdzUserBundle.php`, dans lequel vous mettez un copier-coller de notre `SdzBlogBundle.php` (dans `src/Sdz/BlogBundle/SdzBlogBundle.php`) ;
- Le namespace `Sdz` est déjà enregistré, pas besoin de modifier le `app/autoload.php` ;
- Par contre il faut enregistrer le bundle dans `app/AppKernel.php`.



Pourquoi créer notre propre bundle au lieu de modifier FOSUserBundle directement ? Parce qu'il ne faut **jamais** modifier des fichiers dans le répertoire `vendor/` ! En effet, vous perdriez toutes vos modifications à la prochaine mise à jour du bundle. Et puis ça ne serait pas très beau, tout ce qui nous concerne doit se trouver dans `src/` et pas autre part. Cela prend certes un peu plus de temps, mais c'est nécessaire pour garder un code propre et découplé 😊.

Afin de dire à Symfony2 que notre nouveau bundle SdzUserBundle doit hériter de FOSUserBundle, il faut simplement modifier le `SdzUserBundle.php` comme suit :

**Code : PHP**

```
<?php
// src/Sdz/UserBundle/SdzUserBundle.php

namespace Sdz\UserBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class SdzUserBundle extends Bundle
{
 public function getParent()
 {
 return 'FOSUserBundle';
 }
}
```

Et c'est tout ! On a juste rajouté cette méthode `getParent()`, et Symfony2 va savoir gérer le reste 😊.

### Création de l'entité User

Bon pour l'instant on a beaucoup joué mais on n'a pas fait grand chose hein. Attaquons donc la vraie personnalisation du bundle. La seule chose obligatoire est de créer sa propre entité `User`, l'entité qui va représenter nos membres.

Encore une fois c'est très simple, créez un fichier `User.php` dans le répertoire `src/Sdz/UserBundle/Entity` avec :

**Code : PHP**

```
<?php
// src/Sdz/UserBundle/Entity/User.php

namespace Sdz\UserBundle\Entity;

use FOS\UserBundle\Entity\User as BaseUser;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="sdz_user")
 */
class User extends BaseUser
{
 /**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue(strategy="AUTO")
 */
 protected $id;
}
```

Alors c'est joli, mais pourquoi est-ce que l'on a fait ça ? En fait, le bundle FOSUserBundle ne définit pas vraiment l'entité `User`, il définit une "Mapped Superclass" ! Un nom un peu barbare juste pour dire que c'est une entité abstraite, et qu'il faut en hériter pour faire une vraie entité. C'est donc ce que nous venons juste de faire.

Cela permet en fait de garder la main sur notre entité. On peut ainsi lui rajouter des attributs, en plus de ceux déjà définis. Pour info les attributs qui existent déjà sont :

- `username` : nom d'utilisateur avec lequel l'utilisateur va s'identifier

- email : l'adresse email
- enabled : true ou false suivant que l'inscription de l'utilisateur a été validé ou non (dans le cas d'une confirmation par email par exemple)
- password : le mot de passe de l'utilisateur
- lastLogin : la date de la dernière connexion
- locked : si vous voulez désactiver des comptes
- expired : si vous voulez que les comptes expirent au delà d'une certaine durée

Je vous en passe certains qui sont plus à un usage interne. Sachez tout de même que vous pouvez tous les retrouver dans la définition de la mapped-superclass, qui se trouve ici : [https://github.com/FriendsOfSymfony/FO \[...\] /User.orm.xml](https://github.com/FriendsOfSymfony/FO [...] /User.orm.xml)

Vous pouvez rajouter dès maintenant des attributs à votre entité User, comme vous l'avez fait au chapitre Doctrine2.

### Configuration du bundle

Ensuite, nous devons définir certains paramètres obligatoire au fonctionnement de FOSUserBundle. Ouvrez votre app/config/config.yml et rajoutez la section suivante :

**Code : Autre**

```
app/config/config.yml
#
fos_user:
 db_driver: orm # Le type de BDD à utiliser, nous utilisons l'ORM
 firewall_name: main # Le nom du firewall duquel on utilise l'authentification
 user_class: Sdz\UserBundle\Entity\User # La classe de l'entité User que nous avons créée
```

Et voilà, on a bien installé FOSUserBundle ! Avant d'aller plus loin, créons la table User et ajoutons quelques membres pour les tests.

### Création de la table User

Pour cela rien de plus simple, ressortez votre console et tapez la commande

```
php app/console doctrine:schema:update --force
```

. Et voilà, votre table est créée ! Il existe également une commande fournie par le *bundle* qui permet d'ajouter des utilisateurs, on l'utilise un peu plus bas.

On a fini d'initialiser le *bundle*. Bon bien sûr pour l'instant Symfony2 ne l'utilise pas encore, il manque un peu de configuration, attaquons-la.

## Configuration de la sécurité avec FOSUserBundle

Maintenant on va reprendre tous nos acteurs et les configurer un à un en utilisant le bundle dès que l'on peut. Reprenez le security.yml sous la main, et c'est parti !

### Configuration de la sécurité

#### L'encoder

L'*encoder* est celui qui va hasher les mots de passe avant de les stocker. On utilise couramment la méthode sha512. Dans la configuration, on peut déterminer un *encoder* par interface utilisée pour l'objet User. Définissez donc l'*encoder* suivant :

**Code : Autre**

```
app/config/security.yml
security:
 encoders:
 "FOS\UserBundle\Model\UserInterface": sha512
```

#### Le UserProvider

Je vous l'ai dit également, FOSUserBundle inclut un UserProvider qui va fournir les utilisateurs de type User qu'on a défini plus haut. Pour cela, définissez le UserProvider comme suit :

**Code : Autre**

```
app/config/security.yml
security:
 providers:
 fos_userbundle:
 id: fos_user.user_provider.username
```

Dans cette configuration, "fos\_userbundle" est le nom du provider défini, et "fos\_user.user\_manager" est le nom du service fourni par le bundle FOSUB qui agit comme un provider. Si vous voulez changer de provider, vous devrez donc créer votre

propre service, et changer le nom du service utilisé ici dans la configuration.

Dans la partie du `firewall`, on utilisera donc "fos\_userbundle" comme `UserProvider`.

### Les firewalls

Laissez la sous-partie dev, elle correspond à la désactivation de toute sécurité pour l'accès au profiler (la barre d'information en bas), ainsi qu'aux ressources css, image et js, jusque là tout va bien 😊.

Par contre supprimez les deux autres parties login et secured\_area, on va définir notre propre firewall. L'idée est vraiment simple, copiez-collez les firewalls "login" et "main" suivant :

**Code : Autre**

```
app/config/security.yml

security:

...

firewalls:
 dev:
 pattern: ^/(_(profiler|wdt)|css|images|js)/
 security: false

 # Firewall pour les pages de connexion, inscription, et récupération de
 login:
 pattern: ^/(login$|register|resetting) # Les adresses de ces page
 anonymous: true # On autorise bien évidemn

 # Firewall principal pour le reste de notre site
 main:
 pattern: ^/ # ^/ = tout ce qui commence par /
 form_login: # On définit notre méthode d'authe
 provider: fos_userbundle # On lit l'authentification au prov
 remember_me: true # On active la possibilité du "Se s
 remember_me:
 key: %secret% # On définit la clé pour le rememb
 anonymous: true # On autorise les utilisateurs anor
 logout: true # On autorise la déconnexion manuel


```

Je vous ai déjà expliqué tous les paramètres, et bien commenté le fichier. Ce n'est vraiment que de la syntaxe à connaître, je n'ai pas besoin de vous le détailler plus.

Le choix du `firewall` à appliquer à la requête en cours se fait comme pour le choix des routes : premier arrivé premier servi ! Ainsi, lorsqu'un utilisateur veut accéder à la page /login, le composant Security de Symfony2 va d'abord voir si le `firewall` "dev" correspond, grâce à son paramètre `pattern`. En l'occurrence non. Il passe donc au suivant, le `firewall` "login", et refait le test. Cette fois-ci c'est le bon, il applique les paramètres de ce `firewall` (en l'occurrence, anonymes acceptés). Le `firewall` "main" n'est donc pas utilisé pour cette requête. D'où l'importance de l'ordre des firewalls, gardez-le en tête 😊.

### Configuration de la sécurité : check

Et voilà votre site est prêt à être sécurisé ! En effet on a fini la configuration des `firewalls` et de tous les autres acteurs. Alors évidemment il n'y a pas d'impact sur votre site car pour l'instant on a autorisé les anonymes, donc le `firewall` main ne refuse personne. Essayez de changer le paramètres `anonymous` du `firewall` main en `false`, et regardez l'effet.

Ah oui, pas de route pour /login hein ? C'est déjà une bonne nouvelle, Symfony2 a remarqué qu'on était un utilisateur anonyme, essayant de se connecter à une page qui les refuse. Il nous a redirigé vers la page d'identification, c'est une bonne chose ! Rassurez-vous le formulaire de connexion, comme bien d'autres pages (inscription, récupération de mot de passe, etc.) sont inclus dans le bundle. On a juste à paramétriser les routes.

Ici soyez bien conscient de la transition : nous avons fini de configurer la couche sécurité de Symfony2, mais il nous manque un peu de configuration pour le bundle FOSUserBundle.

## Configuration du bundle FOSUserBundle

### Configuration des routes

En plus de gérer les mécanismes internes, le bundle FOSUserBundle gère aussi les pages classiques comme la page de connexion, celle d'inscription, etc. Pour toutes ces pages, il faut évidemment enregistrer les routes correspondantes. Les développeurs du bundle ont volontairement éclaté toutes les routes dans plusieurs fichiers pour pouvoir personnaliser facilement toutes ces pages. Pour l'instant, on veut juste les rendre disponibles, on les personnalisera plus tard. Rajoutez donc dans votre `app/config/routing.yml` les imports suivant à la suite du notre :

**Code : Autre**

```
app/config/routing.yml

...

fos_user_security:
 resource: "@FOSUserBundle/Resources/config/routing/security.xml"
```

```

fos_user_profile:
 resource: "@FOSUserBundle/Resources/config/routing/profile.xml"
 prefix: /profile

fos_user_register:
 resource: "@FOSUserBundle/Resources/config/routing/registration.xml"
 prefix: /register

fos_user_resetting:
 resource: "@FOSUserBundle/Resources/config/routing/resetting.xml"
 prefix: /resetting

fos_user_change_password:
 resource: "@FOSUserBundle/Resources/config/routing/change_password.xml"
 prefix: /change-password

```

Vous remarquez que les routes sont définies en XML et non en YML comme on en a l'habitude dans ce tutoriel. En effet je vous en avais parlé tout au début, Symfony2 permet d'utiliser plusieurs méthodes pour les fichiers de configuration : YML, XML et même PHP, au choix du développeur. Ouvrez ces fichiers de routes pour voir à quoi ressemblent des routes en XML. C'est quand même moins lisible qu'en YML, c'est pour cela qu'on a choisi YML au début 😊.

Ouvrez vraiment ces fichiers pour connaître les routes qu'ils contiennent. Vous saurez ainsi faire des liens vers toutes les pages que gère le bundle. Inutile de réinventer la roue ! Voici quand même un extrait de la commande

`php app/console router:debug` pour les routes qui concernent ce bundle :

Code : Autre

<code>fos_user_security_login</code>	ANY	/login
<code>fos_user_security_check</code>	ANY	/login_check
<code>fos_user_security_logout</code>	ANY	/logout
<code>fos_user_profile_show</code>	GET	/profile/
<code>fos_user_profile_edit</code>	ANY	/profile/edit
<code>fos_user_registration_register</code>	ANY	/register/
<code>fos_user_registration_check_email</code>	GET	/register/check-email
<code>fos_user_registration_confirm</code>	GET	/register/confirm/{token}
<code>fos_user_registration_confirmed</code>	GET	/register/confirmed
<code>fos_user_resetting_request</code>	GET	/resetting/request
<code>fos_user_resetting_send_email</code>	POST	/resetting/send-email
<code>fos_user_resetting_check_email</code>	GET	/resetting/check-email
<code>fos_user_resetting_reset</code>	GET   POST	/resetting/reset/{token}
<code>fos_user_change_password</code>	GET   POST	/change-password/change-password

## Tester l'authentification

Maintenant que la route pour le formulaire de connexion est configurée, vous pouvez tester l'authentification !

Pour pouvoir la tester, on va ajouter dès maintenant 1 ou 2 utilisateurs. N'oubliez surtout pas PhpMyAdmin, FOSUserBundle dispose de commandes en console qui vont nous faciliter la vie 😊. Exécutez donc la commande

`php app/console fos:user:create` et laissez-vous guider pour ajouter facilement un utilisateur. Vous pouvez recommencer pour en ajouter d'autres. Vous pouvez par contre aller voir le résultat dans la table via PhpMyAdmin, le *bundle* a bien tout rempli, hashé le mot de passe, etc., très pratique !

Maintenant qu'on a des utilisateurs, actualisez l'application... Et voilà ! Le firewall vous a redirigé sur la page /login (sinon vérifiez bien que vous avez mis le paramètre *anonymous* du firewall à *false*), et si vous rentrez le nom d'utilisateur et le mot de passe que vous venez de définir, vous pourrez vous connectez !



Il reste quelques petits détails à gérer comme... la page de login qui n'est pas la plus sexy, sa traduction, et aussi un bouton "Déconnexion" parce que changer manuellement l'adresse en /logout, c'est pas super *user-friendly* !

## Personnalisation esthétique du bundle

Heureusement tout cela est assez simple.



Attention, la personnalisation esthétique que nous allons faire ne concerne en rien la sécurité à proprement parler. Soyez bien conscient de la différence !

### Intégrer le formulaire de connexion dans notre layout

FOSUserBundle utilise un layout volontairement simpliste, parce qu'il a vocation à être écrasé par le nôtre. Le layout actuel est le suivant : [https://github.com/FriendsOfSymfony/FO \[...\] out.html.twig](https://github.com/FriendsOfSymfony/FO [...] out.html.twig)

On va donc tout simplement le remplacer par un template Twig qui va étendre notre layout à nous (qui est dans app/Resources/views/layout.html.twig rappelez-vous). Pour "remplacer" le layout de FOSUB, on va utiliser l'un des avantages d'avoir hérité de ce bundle dans le nôtre. Placez-vous dans notre bundle, dans src/Sdz/UserBundle/Resources/views (créez le répertoire), et créez le fichier layout.html.twig suivant :

Code : HTML & Django

```
{# src/Sdz/UserBundle/Resources/views/layout.html.twig #}

{# On étend notre layout à nous #-}
{%- extends "::layout.html.twig" %}

{# Dans notre layout, il faut définir le block body #}
{%- block body %}

{# On y place le contenu du layout par défaut de FOSUB :
https://github.com/FriendsOfSymfony/FOSUserBundle/blob/master/Resources/views/layout.html.twig #}

{%- if is_granted("IS_AUTHENTICATED_REMEMBERED") %}
 {{ 'layout.logged_in_as'|trans({'%username%': app.user.username})}}
 {% else %}

 {{ 'layout.logout'|trans({}, 'FOSUserBundle') }}

 {% endif %}

{%- for key, message in app.session.getFlashes() %}

{{ message|trans({}, 'FOSUserBundle') }}

{%- endfor %}

{%- block fos_user_content %}
 {%- endblock fos_user_content %}

{%- endblock %}
```

Et voilà, si vous actualisez la page /login (après vous êtes déconnecté via /logout évidemment), vous verrez que le formulaire de connexion est parfaitement intégré dans notre design ! Vous pouvez également tester la page d'inscription sur /register, qui est bien intégrée aussi.



Votre *layout* n'est pas pris en compte ? N'oubliez jamais d'exécuter la commande  
`php app/console cache:clear` lorsque vous avez des erreurs qui vous étonnent !

#### *Traduire les messages*

FOSUB étant un bundle international, le texte est géré par le composant de traduction de Symfony2. Par défaut, celui-ci est désactivé. Pour traduire le texte il suffit donc de l'activer, direction `app/config/config.yml` et décommentez une des premières ligne dans framework :

**Code : Autre**

```
app/config/config.yml

framework:
 translator: { fallback: %locale% }
```

Où `%locale%` est un paramètre défini dans `app/config/parameters.yml`, et que vous pouvez mettre à "fr" si ce n'est pas déjà fait. Ainsi, tous les messages utilisés par FOSUserBundle seront traduits !

#### *Mettre une barre utilisateur*

Pour afficher le nom de l'utilisateur courant, et un petit lien déconnexion, le mieux est de se servir dans le layout de FOSUB. Je vous décortique la structure :

**Code : HTML & Django**

```
{# app/Resources/views/layout.html.twig #}

{# IS_AUTHENTICATED_REMEMBERED correspond à un utilisateur qui s'est identifié en rentrant son mot de passe, ou à un utilisateur qui s'est identifié automatiquement grâce à son cookie remember_me #-}

{# On pourrait utiliser IS_AUTHENTICATED_FULLY qui correspond à un utilisateur qui s'est forcément identifié en rentrant son mot de passe (utile pour les opérations sensibles comme changer d'email #}
```

```
{% if is_granted("IS_AUTHENTICATED_REMEMBERED") %}
 Connecté en tant que {{ app.user.username }} - Déconnexion
{% else %}
 Connexion
{% endif %}
```

Adaptez et mettez ce code dans votre layout, effet garanti 😊.

Pour connaître le nom des routes de FOSUserBundle, je n'ai rien inventé, il faut juste aller dans ses fichiers de routing (dans vendor/bundles/FOS/UserBundle/Resources/config/routing/\*) comme je vous ai dit. Je vous laisse faire toute cette personnalisation, vous avez maintenant toutes les clés en main 😊.

### Gestion des autorisations avec les rôles

Dans cette sous-partie nous allons nous occuper de la deuxième phase de la sécurité : l'autorisation. C'est une phase bien plus simple à gérer heureusement, il suffit juste de demander tel(s) droit(s) à l'utilisateur courant (identifié ou non).

Encore une fois, on va se servir d'un bundle très pratique pour nous simplifier la vie. Celui-ci est déjà intégré dans la distribution standard de Symfony2, il s'agit de [JMSSecurityExtraBundle](#).

### Définition des rôles

Rappelez-vous, on a croisé les rôles dans le fichier security.yml. La notion de rôle et autorisation est très simple, lorsqu'on va limiter l'accès à certains pages, on va se baser sur les rôles de l'utilisateur. Ainsi, limiter l'accès au panel d'administration revient à limiter cet accès aux utilisateurs disposant du rôle ROLE\_ADMIN (par exemple).

Maintenant l'idée est de créer une hiérarchie entre ces rôles. Ainsi, en reprenant nos rôles définis plus haut, on peut sans difficulté dire que le ROLE\_ADMIN comprend les ROLE\_AUTEUR et ROLE\_MODERATEUR. Ainsi, pour limiter l'accès à certaines pages dans notre code, on ne va pas faire "*Si l'utilisateur a ROLE\_AUTEUR ou s'il a ROLE\_ADMIN, alors il peut écrire un article*". Mais grâce à la définition de la hiérarchie, on peut faire simplement "*Si l'utilisateur a ROLE\_AUTEUR*". Car un utilisateur qui dispose du ROLE\_ADMIN dispose également du ROLE\_AUTEUR, c'est une inclusion.

Ce sont ces relations, et uniquement ces relations, que nous allons inscrire dans le fichier app/config/security.yml. Nous avions déjà mis quelques rôles et leurs relations, vous pouvez donc l'adapter aux nouveaux rôles que vous venez de définir. Voici ce que cela donne avec nos quelques rôles d'exemple :

#### Code : Autre

```
app/config/security.yml

security:
 role_hierarchy:
 ROLE_ADMIN: [ROLE_AUTEUR, ROLE_MODERATEUR] # On a dit que l'admin a tous les droits
 ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH] # On garde ce rôle
```

Remarquez que j'ai supprimé le rôle ROLE\_USER, qui n'est pas toujours utile. Avec cette hiérarchie, voici des exemples de tests que l'on peut faire :

- Si l'utilisateur a le rôle ROLE\_AUTEUR, alors il peut écrire un article. Les auteurs et les admins peuvent donc le faire.
- Si l'utilisateur a le rôle ROLE\_ADMIN, alors il peut supprimer un article. Seuls les admins peuvent donc le faire.
- Si l'utilisateur est identifié (via le firewall ou via le rôle spécial IS\_AUTHENTICATED\_REMEMBERED comme on l'a vu), alors il peut lire les articles.

Tous ces tests nous permettront de limiter l'accès à nos différentes pages.

### Tester les rôles de l'utilisateur

L'idée maintenant est de pouvoir tester si l'utilisateur courant dispose de tel ou tel rôle. Ce afin de lui permettre l'accès à la page, ou de lui afficher ou non un certain lien, etc. Libre cours à votre imagination 😊.

Il existe trois méthodes pour faire ce test : les annotations, le service security.context, ou Twig. Ce sont trois façons de faire exactement la même chose.

#### Utiliser directement le service security.context

Ce n'est pas le moyen le plus court, mais c'est celui par lequel passe les deux autres méthodes. Il fallait donc que je vous en parle !

Depuis votre contrôleur ou n'importe quel autre service, il vous faut accéder au service security.context et appeler la méthode isGranted, tout simplement. Par exemple dans notre contrôleur BlogController :

#### Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// ...
// Pensez à rajouter ce use pour l'exception
use
```

```
Symfony\Component\HttpKernel\Exception\AccessDeniedHttpException;

// ...

public function ajouterAction()
{
 // On teste que l'utilisateur dispose bien du rôle
ROLE_AUTEUR
 if(! $this->get('security.context')-
>isGranted('ROLE_AUTEUR'))
 {
 // Sinon on déclenche une exception "Accès Interdit"
 throw new AccessDeniedHttpException('Accès limité aux
auteurs');
 }

 // Ici le code d'ajout d'un article qu'on a déjà fait

 return $this->render('SdzBlogBundle:Blog:ajouter.html.twig',
array(
 'form' => $form->createView(),
));
}
```

C'est tout ! Vous pouvez aller sur [/blog](#), mais impossible d'atteindre la page d'ajout d'un article sur [/blog/ajouter](#), car vous ne disposez pas (encore !) du rôle ROLE\_AUTEUR.

### **Utiliser les annotations dans un contrôleur**

Pour faire exactement ce qu'on vient de faire avec le service security.context, il existe un moyen bien plus rapide et joli : les annotations ! C'est ici qu'intervient le bundle JMSSecurityExtraBundle. Pas besoin d'explication c'est vraiment simple, regardez le code :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

// Plus besoin de rajouter le use de l'exception dans ce cas
// Mais par contre il faut le use pour les annotations du bundle :
use JMS\SecurityExtraBundle\Annotation\Secure;

// ...

/**
 * @Secure(roles="ROLE_AUTEUR")
*/
public function ajouterAction()
{
 // Ici le code d'ajout d'un article qu'on a déjà fait

 return $this->render('SdzBlogBundle:Blog:ajouter.html.twig',
array(
 'form' => $form->createView(),
));
}
```

Et voilà ! Grâce à l'annotation `@Secure`, on a sécurisé notre méthode en une seule ligne, vraiment pratique. Sachez que vous pouvez demander plusieurs rôles en même temps, en faisant `@Secure(roles="ROLE_AUTEUR, ROLE_MODERATEUR")`, qui demandera le rôle AUTEUR **et** le rôle MODERATEUR (ce n'est pas un ou !).

Pour vérifier simplement que l'utilisateur est authentifié, vous pouvez utiliser le rôle spécial `IS_AUTHENTICATED_REMEMBERED`.

Sachez qu'il existe d'autres vérifications possibles avec l'annotation `@Secure`, je vous invite à jeter un oeil à la documentation de [JMSSecurityExtraBundle](#).

### **Depuis une vue Twig**

On l'a déjà croisé plus haut lorsqu'on a affiché soit le bouton déconnexion soit le bouton connexion. Par exemple, à chaque fois que vous avez un lien qui ne doit apparaître que pour certains visiteurs, il faut utiliser la fonction `is_granted()`, comme cela :

**Code : HTML & Django**

```
{% if is_granted('ROLE_AUTEUR') %}
 Ajouter un article
{% endif %}
```

## Gérer les rôles des utilisateurs

Pour l'instant vous n'avez pas pu tester les accès limité car vos utilisateurs n'ont pas de rôle particulier. Pour gérer les rôles il y a deux moyens :

#### Ajouter des rôles depuis la console

Cette méthode est évidemment uniquement à des fin de développement. Ça tombe bien c'est notre cas ! Ouvrez donc la console et exécutez la commande `php app/console fos:user:promote`. Suivez les instructions en mettant un nom d'utilisateur, puis un rôle (testez avec ROLE\_AUTEUR par exemple). Une fois la commande exécutée, il faut vous déconnecter et vous reconnecter afin de prendre en compte les changements. Ensuite, vous pouvez tester que vous avez bien accès à la page d'ajout d'un article 😊.

Sachez au passage qu'il existe quelques autres commandes bien pratiques dans le bundle FOSUserBundle, qui peuvent bien vous faciliter la vie. Elles se trouvent toutes sur la documentation, comme d'habitude : [https://github.com/FriendsOfSymfony/FO \[...\] line\\_tools.md](https://github.com/FriendsOfSymfony/FO [...] line_tools.md)

#### Gérer les rôles depuis le code

L'entité User de FOSUserBundle dispose de méthodes pour gérer les rôles. N'ayez pas peur de jeter un oeil au fichier de l'entité : [https://github.com/FriendsOfSymfony/FO \[...\] odel/User.php](https://github.com/FriendsOfSymfony/FO [...] odel/User.php) Notez particulièrement pour les rôles les méthodes :

- `addRole($role)` pour ajouter un rôle à un utilisateur. Exemple : `$user->addRole('ROLE_AUTEUR');`
- `removeRole($role)` pour supprimer un rôle simplement. Exemple : `$user->removeRole('ROLE_AUTEUR');`
- `getRoles()` pour récupérer les rôles d'un utilisateur. Exemple `$roles = $user->getRoles();`, `$role` est un tableau avec les différent(s) rôle(s).

Ces méthodes vous permettront de gérer les rôles de vos utilisateurs sans passer par la console (qui n'est qu'un moyen de développement).

### Manipuler les utilisateurs

Nous allons voir dans cette dernière sous-partie les moyens pour manipuler vos utilisateurs au quotidien.

#### Manipuler les utilisateurs

Si les utilisateurs sont gérés par FOSUserBundle, ils ne restent que des entités Doctrine2 des plus classiques. Ainsi, vous pourriez très bien vous créer un Repository comme vous savez le faire.

Cependant, profitons du fait que le bundle intègre un UserManager, c'est une sorte de repository avancé. Ainsi, voici les principales manipulations que vous pouvez faire avec :

##### Code : PHP

```
<?php
// Dans un contrôleur :

// Pour récupérer le service UserManager du bundle
$userManager = $this->get('fos_user.user_manager');

// Pour charger un utilisateur
$user = $userManager->findUserBy(array('username' => 'winzou'));

// Pour modifier un utilisateur
$user->setEmail('cetemail@n'existe.pas');
$userManager->updateUser($user); // Pas besoin de faire un flush avec l'entityManager, cette méthode le fait toute seule !

// Pour supprimer un utilisateur
$userManager->deleteUser($user);

// Pour récupérer la liste de tous les utilisateurs
$users = $userManager->findUsers();
```

Si vous avez besoin de plus de fonctions, vous pouvez parfaitement faire un repository perso, et le récupérer comme d'habitude via `<?php $this->getDoctrine()->getEntityManager()->getRepository('SdzBlog:User')`

Et si vous voulez en savoir plus sur ce que fait le bundle dans les coulisses, n'hésitez pas à aller voir le code des contrôleurs du bundle.

### Récupérer l'utilisateur courant

Voici le dernier détail à savoir. Pour récupérer l'utilisateur courant, il faut encore utiliser le service security.context, voici comment :

##### Code : PHP

```
<?php
// Dans un contrôleur

$user = $this->container->get('security.context')->getToken()->getUser();

// Et pour vérifier que l'utilisateur est authentifié (et non un anonyme)
if(! is_object($user))
```

```
{
 throw new AccessDeniedException('Vous n\'êtes pas
authentifié.');
```

N'oubliez pas de vérifier si l'utilisateur est authentifié ou non, sauf si vous êtes sous un firewall qui interdit les anonymes évidemment. Car si l'utilisateur est anonyme, \$user est une chaîne de caractères qui vaut "anon.". Donc si vous essayez de faire \$user->getUsername(), vous aurez évidemment une belle erreur.

Vous avez accès plus facilement à l'utilisateur directement depuis Twig. Vous savez que Twig dispose de quelques variables globales via la variable {{ app }}, et bien l'utilisateur en fait parti, via {{ app.user }} :

Code : Autre

```
Bonjour {{ app.user.username }} - {{ app.user.email }}
```

Encore une fois, attention à ne pas utiliser {{ app.user }} lorsque l'utilisateur n'est pas authentifié, sous peine d'erreur fatale.

Ce chapitre se termine ici pour l'instant. Vous avez maintenant tous les outils en main pour construire votre espace membre, avec un système d'authentification performant et sécurisé, et des accès limités pour vos pages suivant des droits précis.

Sachez que tout ceci n'est qu'une introduction à la sécurité sous Symfony2. Les processus complets sont très puissants mais évidemment plus complexes. Si vous souhaitez aller plus loin pour faire des opérations plus précises, n'hésitez pas à vous référer à la [documentation officielle sur la sécurité](#). Allez jeter un oeil également à la [documentation de FOSUserBundle](#), qui explique comment personnaliser au maximum le bundle, ainsi que [l'utilisation des groupes](#).

Pour information je compte compléter ce chapitre à l'avenir, pour parler des [ACLS](#). C'est un système qui vous permet de définir des droits bien plus finement que les rôles. Par exemple, pour autoriser l'édition d'un article si l'on est admin OU si l'on en est l'auteur. Affaire à suivre !



N'oubliez pas de remettre le paramètre "anonymous" à *true* dans le *firewall*, sauf si vous souhaitez vraiment interdire tous les anonymes sur votre site !

## Les services, théorie et création

Vous avez souvent eu besoin d'exécuter une certaine fonction à plusieurs endroits différents dans votre code ? Comment vérifier une condition sur toutes les pages ? Alors ce chapitre et le prochain sont faits pour vous !

Nous allons découvrir ici une importante fonctionnalité de Symfony : le système de services. Vous le verrez, les services sont utilisés partout dans Symfony2, et sont une fonctionnalité incontournable pour commencer à développer sérieusement un site internet sous Symfony2.

Commençons !

### Qu'est ce qu'un service ?

Un service est simplement un objet PHP qui remplit une fonction.

Cette fonction peut-être simple : envoyer des emails, vérifier qu'un texte n'est pas un spam, etc. Mais elle peut aussi être bien plus complexe : gérer une base de données (le service Doctrine !), etc.

Un service est donc un objet PHP qui a pour vocation d'être accessible depuis n'importe où dans votre code. Pour chaque fonctionnalité dont vous aurez besoin dans toute votre application, vous pourrez créer un ou plusieurs services (et donc une ou plusieurs classes). Il faut vraiment bien comprendre cela : un service est une **simple classe**.

Prenons pour exemple l'envoi d'e-mails. On pourrait alors créer une classe avec comme nom **Mailer** et la définir comme un service. Elle deviendrait alors utilisable n'importe où.

Pour ceux qui connaissent, le concept de service est un bon moyen d'éviter d'utiliser trop souvent à mauvaise escient le pattern Singleton (utiliser une méthode statique pour récupérer l'objet depuis n'importe où).

### La persistance des services

Dans Symfony, chaque service est "**persistant**". Cela signifie simplement que la classe est instanciée une fois (à la première récupération du service) puis cette instance est la même par la suite. Cette persistance permet de manipuler très facilement les services tout au long du processus 😊.

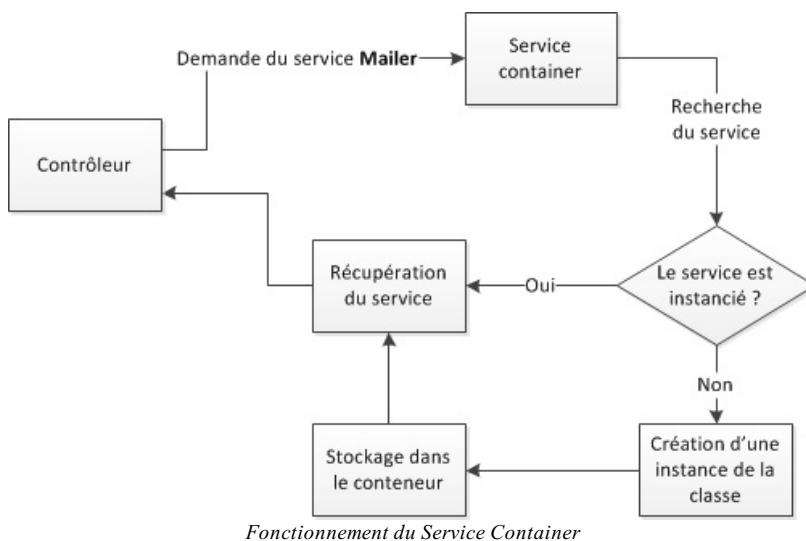
### Le service container

 Mais alors, si un service est juste une classe, pourquoi appeler celle-ci un service ? Et pourquoi utiliser les services ?

L'intérêt réel des services réside dans leurs associations avec un "conteneur de services". Ce conteneur de services ("*services container*" en anglais) est une sorte de super-objet qui gère tous les services. Ainsi, pour accéder à un service, il faut passer par le conteneur.

L'intérêt principal du conteneur est de pouvoir utiliser une classe sans connaître son nom. Par exemple : je cherche à envoyer un e-mail. Je demande au *service container* de me retourner le service nommé "Mailer", bien que je ne connaisse pas quelle classe est derrière ce nom. Le conteneur me renvoie une instance de la classe associée (*SwiftMailer* par exemple), que je peux utiliser pour envoyer mon e-mail. Cela permet de pouvoir changer de classe pour un service donné, sans pour autant changer le code de tous les endroits où le service est utilisé.

Voilà un schéma explicatif du positionnement du conteneur de services :



Dans Symfony le *ServiceContainer* est vraiment un élément central, d'où ce chapitre !

### En pratique

Continuons sur notre exemple d'e-mail. Dans Symfony, il existe un composant appelé *Swiftmailer*, qui permet d'envoyer des e-mails simplement. Il est présent par défaut dans Symfony, sous forme de service.

Pour accéder à un service déjà enregistré, il suffit d'utiliser la méthode `get($identifiantService)` du conteneur. Par exemple :

**Code : PHP**

```
<?php
$container->get('mailer');
```

Pour avoir la liste des services disponibles, utilisez la commande

**Code : Console**

```
php app/console container:debug
```

Et comment j'accède à \$container moi ?!



En effet, la question est importante. Dans Symfony il existe un classe nommée *ContainerAware* qui possède un attribut **\$container**. Le cœur de Symfony alimente ainsi les classes du framework en utilisant la méthode `setContainer()`. Donc pour toute classe de Symfony héritant de *ContainerAware*, on peut faire ceci :

**Code : PHP**

```
<?php
$this->container->get('mailer');
```

Heureusement pour nous, la classe de base des contrôleurs nommée *Controller* hérite de cette classe *ContainerAware*, on peut appliquer ceci aux contrôleurs :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller; // Cette
classe étend ContainerAware

class BlogController extends Controller
{
 public function indexAction()
 {
 $swiftmailer = $this->container->get('mailer'); // On a donc
accès au conteneur

 var_dump($swiftmailer);
 exit;

 return $this->render('SdzBlogBundle:Blog:index.html.twig');
 }
}
```

Il se peut que vous ayez déjà rencontré, depuis un contrôleur, l'utilisation de `get()` sans passer par l'attribut `container`, comme ceci : `$this->get('mailer')`. C'est parce que la classe *Controller* fournit un raccourci, la méthode `$this->get()` faisant simplement appel à la méthode `$this->container->get()`.

Donc dans un contrôleur, `$this->get(...)` est strictement équivalent à `$this->container->get(...)`.

**Créer un service****Création de la classe du service**

Maintenant que nous savons utiliser un service, apprenons à le créer. Comme un service n'est qu'une classe, il suffit de créer un fichier n'importe où et de créer une classe dedans.

La seule convention à respecter, de façon générale dans Symfony, c'est de mettre notre classe dans un namespace correspondant au dossier où est le fichier. Par exemple, la classe `Sdz\BlogBundle\Service\SdzAntispam` doit se trouver dans le répertoire `src/Sdz/BlogBundle/Service/SdzAntispam.php`. C'est ce que nous faisons depuis le début du tutoriel 😊.

Je vous propose, pour suivre notre fil rouge du blog, de créer un système anti-spam. Notre besoin : détecter les spams à partir d'un simple texte. Comme c'est une fonction à part entière, et qu'on aura besoin d'elle à plusieurs endroits (pour les articles et pour les commentaires), faisons-en un service. Ce service devra être réutilisable simplement dans d'autre projets Symfony : il ne devra pas être dépendant d'un élément de notre blog. Je nommerais ce service **SdzAntispam**, mais vous pouvez le nommer

comme vous le souhaitez. Il n'y a pas de règle précise à ce niveau, mise à part que l'utilisation des underscores (\_) est fortement déconseillée.

Si votre service est relativement indépendant (il n'y a pas d'autres services autour du même thème), il est bon de mettre ce service dans un dossier **Service** de votre bundle, mais à vrai dire vous pouvez faire comme vous le souhaitez.

Créons donc le fichier `src/Sdz/BlogBundle/Service/SdzAntispam.php`, avec ce code pour l'instant :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Service/SdzAntispam.php

namespace Sdz\BlogBundle\Service;

/**
 * Un anti-spam simple pour Symfony2.
 *
 * @author Leglopin
 */
class SdzAntispam
{

}
```

C'est tout ce qu'il faut pour avoir un service. Il n'y a vraiment rien d'obligatoire, vous y mettez ce que vous voulez. Pour l'exemple, faisons un rapide anti-spam : considérons qu'un message est un spam s'il contient au moins 3 liens ou adresse e-mail. Voici ce que j'obtiens :

**Code : PHP**

```
<?php
// src/Sdz/BlogBundle/Service/SdzAntispam.php

namespace Sdz\BlogBundle\Service;

/**
 * Un anti-spam simple pour Symfony2.
 *
 * @author Leglopin
 */
class SdzAntispam
{
 /**
 * Vérifie si le texte est un spam ou non.
 * Un texte est considéré comme spam à partir de 3 liens
 * ou adresses e-mails dans son contenu.
 *
 * @param string $text
 */
 public function isSpam($text)
 {
 if(($this->countLinks($text) + $this->countMails($text)) >=
3)
 {
 return true;
 }
 else
 {
 return false;
 }
 }

 /**
 * Compte les URL de $text.
 *
 * @param string $text
 */
 private function countLinks($text)
 {
 preg_match_all(
 '#(http|https|ftp)://([A-Z0-9][A-Z0-9_-]*(:.[A-Z0-9][A-
Z0-9_-]*)+):?(d+)?/?#i',
 $text,
 $matches);
 return count($matches[0]);
 }

 /**
 * Compte les e-mails de $text.
 *
 * @param string $text
 */
 private function countMails($text)
 {
```

```

preg_match_all(
 '#[a-z0-9._-]+@[a-z0-9._-]{2,}\.[a-z]{2,4}#i',
 $text,
 $matches);

return count($matches[0]);
}
}

```

## Création de la configuration du service

Maintenant que nous avons créé notre service, il faut le signaler à Symfony. Un service se définit par sa classe ainsi que sa configuration. Pour cela, nous pouvons utiliser le fichier **src/Sdz/BlogBundle/Resources/config/services.yml**.

Si vous avez généré votre bundle avec le generator en répondant "oui" pour créer toute la structure du bundle, alors ce fichier est chargé automatiquement. Vérifiez-le en confirmant que le répertoire `DependencyInjection` de votre bundle existe, il devrait contenir le fichier `SdzBlogExtension.php`.

Si ce n'est pas le cas, vous devez créer le fichier `DependencyInjection/SdzBlogExtension.php` (adaptez à votre bundle évidemment) avec le contenu suivant :

**Code : PHP**

```

<?php

namespace Sdz\BlogBundle\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\Loader;

class SdzBlogExtension extends Extension
{
 public function load(array $configs, ContainerBuilder $container)
 {
 $loader = new Loader\YamlFileLoader($container, new
FileLocator(__DIR__.'/../Resources/config'));
 $loader->load('services.yml');
 }
}

```



Ouvrez ou créez le fichier `Ressources/config/services.yml` de votre bundle, et ajoutez-y la configuration pour notre service :

**Code : Autre**

```

services:
 sdz_blog.antspam:
 class: Sdz\BlogBundle\Service\SdzAntispam

```

Dans cette configuration :

- "sdz\_blog.antspam" est le nom de notre service fraîchement créé. De cette manière, le service sera accessible via `<?php $container->get('sdz_blog.antspam');`. Essayez de respecter cette convention de préfixer le nom de vos services par "nomApplication\_nomBundle". Pour notre bundle `Sdz\BlogBundle`, on a donc préfixé notre service de "sdz\_blog".
- "class" est un attribut obligatoire de notre service, il définit simplement le namespace complet de la classe du service.

Il existe bien sûr d'autres attributs pour affiner la définition de notre service, nous les verrons dans le prochain chapitre.

Sachez également que le conteneur de Symfony2 permet de stocker aussi bien des services (des classes) que des paramètres (des variables).

Pour définir un paramètre, la technique est la même que pour un service, dans le fichier `services.yml` :

**Code : Autre**

```

parameters:
 mon_parametre: ma_valeur

services:
 #

```



Et pour accéder à ce paramètre, la technique est la même également, sauf qu'il faut utiliser la méthode `<?php $container->getParameter($identifiant);` au lieu de `get()`.

## Utilisation du service

Maintenant que notre classe est définie, et notre configuration déclarée, nous pouvons nous servir du service. Voici un exemple simple de l'utilisation que l'on pourrait en faire :

Code : PHP

```
<?php
// src/Sdz/BlogBundle/Controller/BlogController.php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
 public function indexAction()
 {
 $antispam = $this->container->get('sdz_blog.antispam');

 if($antispam->isSpam($text))
 {
 exit('Votre message a été détecté comme spam !');
 }

 return $this->render('SdzBlogBundle:Blog:index.html.twig');
 }
}
```

Et voilà, vous avez créé et utilisé votre premier service !

Les services sont extrêmement utilisés dans Symfony2. On n'a pas encore vu tout leur potentiel jusque maintenant, mais ils auront un aspect beaucoup plus intéressant au prochain chapitre, où nous allons utiliser les services pour modifier le comportement de Symfony2 !

Afin de bien comprendre les mécanismes vu dans ce chapitre, je vous conseille de lire ces deux tutoriels :

- [Introduction à l'injection de dépendances en PHP](#) de vincent1870
- [Les designs patterns : l'injection de dépendance](#) de vyk12

Ces deux tutoriels présentent l'injection de dépendance (l'injection de dépendance est liée au principe de service) de manière concise. Cependant, comme vous pourrez le voir au prochain chapitre, Symfony va bien plus loin 😊.

Tout ne fait que commencer !

## Les services, utilisation poussée

Cette deuxième partie est réservé aux fonctionnalités les plus intéressantes des services. Maintenant que les bases vous sont acquises, nous allons pouvoir découvrir des fonctionnalités très puissantes de Symfony.

### Les arguments

La plupart du temps, vos services ne fonctionneront pas seuls, et vont nécessiter l'utilisation d'autres services, de paramètres ou de variables. Il a donc fallu trouver un moyen propre et efficace pour pallier à ce problème. Pour passer des arguments à votre service, il faut utiliser le fichier de configuration :

#### Code : Autre

```
parameters:
services:
 sdz_blog_antispam:
 class: Sdz\BlogBundle\Service\SdzAntispam
 arguments: [] # Tableau d'arguments
```

Les arguments peuvent être :

- Des valeurs normales en YAML (des booléens, des chaînes de caractères, des nombres, etc.)
- Des paramètres (définis dans le parameters.ini par exemple) : l'identifiant du paramètre est encadré de signes % : %parametre\_id%
- Des services : l'identifiant du service est précédé d'un arrobase : @service\_id

Par exemple :

#### Code : Autre

```
parameters:
services:
 sdz_blog_antispam:
 class: Sdz\BlogBundle\Service\SdzAntispam
 arguments: [@doctrine, %locale%, 3]
```

Dans cet exemple, notre service utilise

- @doctrine : le service Doctrine (pour utiliser la base de données),
- %locale% : le paramètre locale (pour récupérer la langue),
- 3 : et le nombre 3 (qu'importe son utilité !).

Une fois vos arguments définis dans la configuration, il vous suffit de les récupérer avec votre constructeur. Les arguments de la configuration et ceux du constructeur vont donc de paire. Si vous modifiez l'un, n'oubliez pas d'adapter l'autre. Voici donc le constructeur adapté à notre nouvelle configuration :

#### Code : PHP

```
<?php
// src/Sdz/BlogBundle/Service/SdzAntispam.php

namespace Sdz\BlogBundle\Service;

use Doctrine\Bundle\DoctrineBundle\Registry;

/**
 * Un anti-spam simple pour Symfony2.
 *
 * @author Leglopin
 */
class SdzAntispam
{
 /**
 * @var Symfony\Bundle\DoctrineBundle\Registry
 */
 protected $doctrine;

 protected $locale;

 protected $nbFoundedForSpam;

 public function __construct(Registry $doctrine, $locale,
 $nbFoundedForSpam)
 {
 $this->doctrine = $doctrine;
 $this->locale = (string) $locale;
 $this->nbFoundedForSpam = (int) $nbFoundedForSpam;
 }
}
```

```

 }

 public function isSpam($text)
 {
 if (($this->countLinks($text) + $this->countMails($text)) >=
 $this->nbFoundedForSpam) // Nous utilisons ici l'argument
 $nbFoundedForSpam
 {
 return true;
 }
 else
 {
 return false;
 }
 }

 // ...
}

```

 Vous pouvez remarquer que les arguments permettent de transmettre des services à d'autres services : c'est pour cette raison qu'on appelle ce concept **l'injection de dépendance** (*dependency injection* en anglais). Au lieu de faire appel directement dans le code à des classes, on fait appel à un conteneur de services qui crée alors des **dépendances dynamiques** 😊. La boucle est bouclée !

## Les calls

Cette fonctionnalité est assez simple : elle permet d'appeler directement après la construction du service certaines fonctions.

Ainsi, pour faire exactement la même chose que précédemment, vous pouvez configurer votre service comme ceci :

**Code : Autre**

```

parameters:

services:
 sdz_blog.antispam:
 class: Sdz\BlogBundle\Service\SdzAntispam
 arguments: [@doctrine, 3]
 calls:
 - [setLocale, [%locale%]]

```

Vous pouvez définir plusieurs calls, en rajoutant des lignes à tiret. Le premier argument (ici, setLocale) est le nom de la méthode à exécuter. Le deuxième argument (ici, [ %locale% ]) est le tableau des arguments à transmettre à la méthode exécutée.

En adaptant bien sûr le service comme ceci :

**Code : PHP**

```

<?php
// src/Sdz/BlogBundle/Service/SdzAntispam.php

namespace Sdz\BlogBundle\Service;

/**
 * Un anti-spam simple pour Symfony2.
 *
 * @author Leglopin
 */
class SdzAntispam
{
 protected $doctrine;

 protected $locale;

 protected $nbFoundedForSpam;

 public function __construct(Registry $doctrine,
 $nbFoundedForSpam)
 {
 $this->doctrine = $doctrine;
 $this->nbFoundedForSpam = $nbFoundedForSpam;
 }

 // C'est cette méthode qui va être exécutée par le call
 public function setLocale($locale)
 {
 $this->locale = $locale;
 }

 // ...
}

```

L'utilité des calls est surtout remarquable pour l'intégration des libraires externes (Zend Framework, Geshi, etc.).

## Les tags

Une fonctionnalité très importante des services est la possibilité d'utiliser les **tags**. Les tags sont en fait des options ajoutées à un service pour lui donner la capacité d'intervenir différemment avec Symfony. Par exemple, il existe un tag **twig.extension** qui signale à Symfony que le service est une extension Twig.

Nous allons ici parler des tags les plus importants, ainsi que comment les mettre en place.

## Extension Twig

Pour l'exemple, nous allons transformer notre SdzAntispam pour lui ajouter la possibilité de vérifier un texte de la vue. Adaptons la configuration du service :

### Code : Autre

```
parameters:
services:
 sdz_blog.antispam:
 class: Sdz\BlogBundle\Service\SdzAntispam
 tags:
 - { name: twig.extension }
```

Notre service sera donc traité comme une extension Twig. Cependant, un service implantant ce tag doit étendre `Twig_Extension`. Adaptons le code du service en implémentant les méthodes `getFunctions()` et `getName()` obligatoire :

### Code : PHP

```
<?php
// src/Sdz/BlogBundle/Service/SdzAntispam.php

namespace Sdz\BlogBundle\Service;

/**
 * Un anti-spam simple pour Symfony2.
 *
 * @author Leglopin
 */
class SdzAntispam extends \Twig_Extension
{
 // La méthode getName(), obligatoire
 public function getName()
 {
 return 'SdzAntispam';
 }

 // La méthode getFunctions(), qui retourne un tableau avec les
 // fonctions qui peuvent être appelées depuis cette extension
 public function getFunctions()
 {
 return array(
 'antispam_check' => new \Twig_Function_Method($this,
 'isSpam')
);
 }

 // 'antispam_check' est le nom de la fonction qui sera
 // disponible sous Twig
 // 'new \Twig_Function_Method($this, 'isSpam')' est la
 // façon de dire que cette fonction va exécuter notre méthode isSpam
 // ci-dessous
}

/**
 * Vérifie si le texte est un spam ou non.
 * Un texte est considéré comme spam à partir de 3 liens
 * ou adresses e-mails dans son contenu.
 *
 * @param string $text
 */
public function isSpam($text)
{
 if(($this->countLinks($text) + $this->countMails($text)) >=
 3)
 {
 return true;
 }
 else
 {
 return false;
 }
}
```

```

 }
 // ...
}
```

Vous pouvez maintenant utiliser la fonction dans Twig :

**Code : Autre**

```
{ { antispam_check('Texte') } }
```

Pour plus d'informations à propos de la création d'extensions pour Twig, lisez ceci :  
<http://twig.sensiolabs.org/doc/extensions.html>.

## Les évènements du cœur

Les services peuvent être utilisés avec le gestionnaire d'évènements. Un chapitre sera bientôt disponible à ce sujet. Vous pouvez en attendant lire [la documentation à ce sujet](#).

## Les types de champs de formulaire

Cette fonctionnalité est très pratique, mais comme elle n'est pas documentée, je ne connais que ce que j'ai découvert par moi-même, et il se peut qu'il manque quelques fonctionnalités qui pourraient être intéressantes. N'hésitez pas à faire des recherches .

Pour déclarer un type de champ, il faut utiliser le tag **form.type** et indiquer un alias. Par exemple, pour un champ **ckeditor** (un éditeur WYSIWYG) :

**Code : Autre**

```

services:
 sdz_blog_ckeditor:
 class: Sdz\BlogBundle\Form\Extension\CkeditorType
 tags:
 - { name: form.type, alias: ckeditor }
```

La classe CkeditorType contient alors :

**Code : PHP**

```

<?php
// src/Sdz/BlogBundle/Form/Extension/CkeditorType.php

/**
 * Type de champ de formulaire pour CKEditor.
 *
 * @author Leglopin
 */
namespace Sdz\BlogBundle\Form\Extension;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class CkeditorType extends AbstractType
{
 public function getParent()
 {
 return 'textarea';
 }

 public function getName()
 {
 return 'ckeditor';
 }

 public function setDefaultOptions(OptionsResolverInterface
$resolver)
 {
 $resolver->setDefaults(array(
 'attr' => array('class' => 'ckeditor')
));
 }
}
```

Vous venez de déclarer le type `ckeditor` (nom de l'alias). Ce type hérite de toutes les fonctionnalités d'un textarea (grâce à la méthode `getParent()`) tout en disposant de la classe CSS `ckeditor` (définie dans la méthode `setDefaults()`) vous permettant, en ajoutant CKEditor à votre site, de transformer vos textarea en éditeur WYSIWYG. Pour l'utiliser, modifiez vos `FormType` pour utiliser 'ckeditor' à la place de 'textarea'. Par exemple, dans notre `ArticleType` :

#### Code : PHP

```
<?php
namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ArticleType extends AbstractType
{
 public function buildForm(FormBuilderInterface $builder, array
$options)
 {
 $builder
 ->add('date', 'date')
->add('titre', 'text')
 ->add('contenu', 'ckeditor')
 ->add('auteur', 'text')
 ->add('publication', 'checkbox', array('required' => false))
 }

 public function getName()
 {
 return 'sdz_blogbundle_articletype';
 }

 public function setDefaultOptions(OptionsResolverInterface
$resolver)
 {
 $resolver->setDefaults(array(
 'data_class' => 'Sdz\BlogBundle\Entity\Article'
));
 }
}
```

Pour en savoir plus, vous pouvez regarder le fichier [vendor\symfony\src\Symfony\Component\Form\AbstractType.php](#).

## Les autres tags

Il existe beaucoup d'autres tags. Vous pouvez trouver la liste des tags par défaut dans Symfony ici : [http://symfony.com/doc/2.0/reference/dic\\_tags.html](http://symfony.com/doc/2.0/reference/dic_tags.html), mais il faut savoir que cette liste n'est pas complète car d'autres bundles ajoutent leur propre tags 😊.

La liste officielle n'étant pas très pratique pour comprendre comment utiliser certains tags, je compte faire ici une liste avec une description plus détaillée de chacun d'entre eux 😊 :

Identifiant	Description	Exemple
translation.loader	Utilise votre service en tant que classe de chargement de traductions, sur le même principe que <code>routing.loader</code> . Cette fonctionnalité permet d'ajouter le support de plus de formats de configuration de traductions. Par exemple, il existe un loader <code>YamlFileLoader</code> . Pour déclarer votre classe de chargement, utilisez le tag <code>translation.loader</code> avec un alias qui doit correspondre à l'extension que peut comprendre votre loader. Votre classe de chargement doit correspondre à l'interface <code>LoaderInterface</code> .	<p><b>Code : Autre</b></p> <pre>services:     routing.loader.your_loader_name:         class: Fully\Qualified\Loader\Class\Name         tags:             -         { name: translation.loader, alias: ini }</pre>
		<p><b>Code : Autre</b></p> <pre>services:     routing.loader.your_loader_name:         class: Fully\Qualified\Loader\Class\Name         tags:</pre>

		<pre>- { name: routing.loader }</pre> <p>Interface : <a href="#">Secret (cliquez pour afficher)</a></p> <p><b>Code : PHP</b></p> <pre>&lt;?php interface RouterLoader {     /**      * Charge les routes depuis une      * ressource.      * Cette méthode doit retourner une      * collection RouteCollection.      *      * @param mixed \$string La ressource      * @param string \$type Le type de      * ressource      *      * @return RouteCollection      *      * @api      */     public function load(\$string,     \$type = null);      /**      * Cette méthode est utilisée pour      * savoir si le Loader est capable de      * charger la ressource.      *      * @param mixed \$resource La ressource      * @param string \$type Le type de      * ressource      *      * @return Boolean Retourne TRUE si il      * est possible d'utiliser ce Loader,      * FALSE sinon      *      * @api      */     public function supports(\$string,     \$type = null); }</pre>
routing.loader	<p>Utilise votre service en tant que classe de chargement de routes. Cette fonctionnalité permet d'ajouter le support de plus de formats de configuration de routes. Par exemple, il existe des loaders <code>PhpFileLoader</code>, <code>XmFileLoader</code> et <code>YamlFileLoader</code>. Pour déclarer votre classe de chargement, utilisez le tag <code>routing.loader</code>.</p> <p>Votre classe de chargement doit correspondre à l'interface ci-contre :</p>	
templating.helper	<p>Utilise votre service en tant que Helper de vue. Un helper de vue est différent d'une extension Twig car il n'est utilisable que dans les templates PHP. Vous devez définir un alias pour utiliser ce dernier dans votre vue.</p> <p>Une fois déclaré, vous pouvez accéder à votre helper ainsi : <code>&lt;?php \$view['alias_name'] ; ?&gt;</code>, soit par exemple pour <code>SdzAntispam</code> : <code>&lt;?php \$view['antispam'] -&gt;isSpam(/.../); ?&gt;</code></p>	<p><b>Code : Autre</b></p> <pre>services:     templating.helper.your_helper_name:         class: Fully\Qualified\Helper\Class\Name         tags:             -         { name: templating.helper, alias: alias_name }</pre>
templating.engine	<p>Ce tag indique à Symfony que votre classe permet d'utiliser un nouveau moteur de templates. Par exemple, Twig possède sa classe <code>TwigEngine</code>. Pour déclarer un moteur de templates, utilisez le tag <code>templating.engine</code>.</p> <p>Vous devez ensuite créer votre classe <code>Engine</code> sur le modèle de l'interface <code>EngineInterface</code></p>	<p><b>Code : Autre</b></p> <pre>services:     templating.engine.your_engine_name:         class: Fully\Qualified\Engine\Class\Name         tags:             -         { name: templating.engine }</pre>

## Les services courants de Symfony2

### Les services courants de Symfony

Certains d'entre vous ont peut-être déjà remarqué qu'ils avaient déjà utilisé des services auparavant. En effet, Symfony fournit par défaut pas mal de services, dont appartiennent par exemple Doctrine, Request ou Swiftmailer. Je vous propose d'étudier plus en détails certains services qui pourront vous être très utiles par la suite !

Identifiant	Description
doctrine.orm.entity_manager	Ce service est l'instance de l'EntityManager de Doctrine ORM. Vous avez probablement déjà utilisé Doctrine en tant que service, mais peut-être sans le savoir car la classe Controller définit une méthode appelée <code>getDoctrine()</code> qui renvoi le service directement. Récupérer l' <code>EntityManager</code> dans un service est extrêmement pratique pour ensuite effectuer des opérations en base de données, récupérer un <code>repository</code> , etc.
event_dispatcher	Ce service donne accès à l'évent dispatcher. Cet élément est un peu trop complexe pour le réduire à ces quelques lignes, mais disons que c'est un objet pour gérer les événements et les fonctions qui réagissent à ces événements. Un chapitre sur le gestionnaire d'événements est en cours de rédaction 😊.
kernel	Ce service vous donne accès au coeur de Symfony. Grâce à lui, vous pouvez localiser des bundles, récupérer le chemin de base du site, etc. Vöyez le fichier <code>Kernel.php</code> pour connaître toutes les possibilités.
logger	Ce service est la classe de logs. Grâce à lui, vous pouvez utiliser des fichiers de logs très simplement. Symfony utilise Monolog pour gérer ses logs. Vöyez les fichiers <code>vendor\symfony\src\Symfony\Bridge\Monolog\Logger.php</code> et <code>vendor\monolog\src\Monolog\Logger.php</code> pour en savoir plus.
mailer	Ce service vous renvoi par défaut une instance de SwiftMailer, vous permettant d'envoyer des e-mails. Vous pouvez consulter la documentation de <code>SwiftMailer</code> pour en savoir plus.
request	<p>Ce service est très important : il vous donne une instance de <code>Request</code> qui représente la requête du client. Vous pouvez par exemple :</p> <ul style="list-style-type: none"> <li>Récupérer la session en cours : <code>&lt;?php \$this-&gt;get('request')-&gt;getSession()</code></li> <li>Récupérer l'IP du client : <code>&lt;?php \$this-&gt;get('request')-&gt;getClientIp()</code></li> <li>Récupérer la méthode de la requête (comme dans les formulaires) : <code>&lt;?php \$this-&gt;get('request')-&gt;getMethod()</code> : <b>POST</b>, <b>GET</b>, <b>PUT</b>, etc.</li> <li>Savoir si c'est une requête en AJAX : <code>&lt;?php \$this-&gt;get('request')-&gt;isXmlHttpRequest()</code></li> <li>Et bien d'autres choses ...</li> </ul> <p>Je vous laisse découvrir tout ceci : <a href="#">Symfony\Component\HttpFoundation\Request.php</a></p>
router	<p>Ce service vous donne accès au routeur (<a href="#">Symfony\Component\Routing\Router</a>). C'est cet objet qui génère vos routes et qui les transforme en URL. Vous pouvez par exemple générer une route :</p> <p><b>Code : PHP</b></p> <pre>&lt;?php \$this-&gt;get('router')-&gt;generate('homepage');</pre> <p>Ou trouver une route correspondant à une URL :</p> <p><b>Code : PHP</b></p> <pre>&lt;?php \$this-&gt;get('router')-&gt;match(\$url);</pre> <p> La classe Controller a aussi un raccourci pour générer une route : vous pouvez générer une URL dans un contrôleur avec <code>&lt;?php \$this-&gt;generateUrl(\$nomRoute, \$parametres, \$estAbsolu);</code></p>
security.context	<p>Ce service est surtout utile car il permet de récupérer l'utilisateur courant :</p> <p><b>Code : PHP</b></p> <pre>&lt;?php \$this-&gt;get('security.context')-&gt;getToken()-&gt;getUser();</pre> <p> Ce code fonctionne seulement si l'utilisateur est connecté. Pour le vérifier, il faut vérifier que <code>&lt;?php \$this-&gt;get('security.context')-&gt;getToken();</code> n'est pas nul.</p>
service_container	Ce service vous renvoi le service container. Comme vous pouvez vous en douter, on ne l'utilise pas dans les contrôleurs. Il est surtout utile dans les autres services, où on le passe en argument pour accéder au conteneur 😊.
session	Ce service représente les sessions. Vöyez le fichier <a href="#">Symfony\Component\HttpFoundation\Session\Session.php</a> pour en savoir plus.
twig	Ce service représente une instance de <a href="#">Twig_Environment</a> . Il permet d'afficher une vue. Vous pouvez en savoir plus en lisant la documentation de <a href="#">Twig</a> . Ce service peut être utile pour modifier l'environnement de Twig depuis l'extérieur (lui ajouter des extensions, etc.).

templating

Ce service représente le moteur de templates. Cela peut être Twig, PHP ou tout autre moteur utilisable. Ce service montre l'intérêt de l'injection de dépendances : on réussit à faire un code valide pour plusieurs moteurs de templates en même temps, et changer de moteur ne nécessite pas de modifier le code.

La classe *Controller* fait référence à ce service pour afficher une vue. Quand vous utilisez la méthode `$this->render()` d'un contrôleur, vous utilisez en fait le service templating. Voyez vous-même le contenu de cette méthode `render()` :

Code : PHP

```
<?php
//
// vendor\symfony\src\Symfony\Bundle\FrameworkBundle\Controller\Controller.php

/**
 * Renders a view.
 *
 * @param string $view The view name
 * @param array $parameters An array of parameters to pass to the view
 * @param Response $response A response instance
 *
 * @return Response A Response instance
 */
public function render($view, array $parameters = array(), Response
$response = null)
{
 return $this->container->get('templating')->renderResponse($view,
$parameters, $response);
}
```



J'espère avoir réussi à vous éclairer à propos de la gestion des services dans Symfony.

Vous pouvez maintenant vous rendre compte que grâce à eux, de nouvelles possibilités s'offrent à vous : les extensions Twig, l'intégration de bibliothèque tierces, etc.



Ces deux chapitres sur les services ont été écrits à l'origine par Titouan Galopin ([Leglopis](#)).

## Partie 5 : Astuces et points particuliers

Cette partie recense différentes astuces et points particuliers de Symfony2, qui vous permettent de réaliser des choses précises dans votre projet.

Comme ces petites et moyennes astuces les couvrent souvent plusieurs notions en même temps, il est impossible de les intégrer dans le cours du tutoriel, c'est pourquoi une partie leur est dédiée. Je vous invite donc à en lire une de temps en temps, car ce sont souvent des points très pratiques à connaître !

Bon code.

### Récupérer directement des entités Doctrine dans son contrôleur

L'objectif de cette astuce, comme de beaucoup d'autres, est de vous faire gagner du temps et des lignes de code. Sympa, non ?



#### Théorie : pourquoi un ParamConverter ?

#### Récupérer des objets Doctrine avant même le contrôleur

Sur la page d'affichage d'un article de blog, par exemple, n'êtes-vous pas fatigué de toujours devoir vérifier l'existence de l'article demandé et de l'instancier vous-même ? N'avez-vous pas l'impression d'écrire toujours et encore les mêmes lignes ?

Code : PHP

```
<?php
public function voirAction($id)
{
 if(! $article = $this->get('doctrine')->getEntityManager()-
>getRepository('Sdz\BlogBundle\Entity\Article')->find($id))
 {
 throw new NotFoundHttpException(sprintf('L\'article
id:"%s" n\'existe pas.', $id));
 }

 // votre vrai code

 return $this->render('SdzBlogBundle:Blog:voir.html.twig',
array('article' => $article));
}
```

Pour enfin vous concentrer sur votre code métier, Symfony a évidemment tout prévu !

### Les ParamConverters

Vous pouvez créer ou utiliser des **ParamConverters** qui vont agir juste après le routeur. Les **ParamConverters** vont, comme leur nom l'indique, **convertir** les paramètres de votre route au format que vous préférez.

En effet, depuis la route, vous ne pouvez pas tellement agir sur vos paramètres. Tout au plus, vous pouvez leur imposer des contraintes via une *regex*. Les **ParamConverter** pallient cette limitation en agissant après le routeur.

### Un ParamConverter utile : DoctrineParamConverter

Vous l'aurez deviné, ce **ParamConverter** va nous convertir nos paramètres directement en *Entity* Doctrine !

- L'idée : dans le contrôleur, à la place de la variable `<?php $id`, on souhaite récupérer directement une variable `<?php $article` qui correspond à l'article portant l'id `<?php $id`.
- Le bonus : on veut également que, si l'article portant l'id `<?php $id` n'existe pas, une exception 404 soit levée. Après tout, c'est comme si l'on mettait dans la route : "requirements: Article exists" !

### Un peu de théorie sur les ParamConverter

Comment fonctionne un **ParamConverter** ?

Ce n'est en fait qu'un simple *listener* qui écoute l'événement **kernel.controller**. Cet événement est déclenché lorsque le contrôleur sait quel contrôleur appeler (après le routeur, donc), mais avant d'exécuter effectivement le contrôleur. Ainsi, lors de cet événement, le **ParamConverter** va lire la méthode de votre contrôleur pour trouver soit l'annotation, soit le type de variable que vous souhaitez. Fort de ces informations, il va se permettre de modifier le paramètre de la requête (il y a accès). Ainsi, depuis votre contrôleur, vous n'avez plus le paramètre original tel que défini dans la route, mais un paramètre modifié par votre **ParamConverter** qui s'est exécuté avant votre contrôleur.

### Pratique : utilisation de DoctrineParamConverter

#### Utiliser DoctrineParamConverter

Ce **ParamConverter** fait partie du *bundle* `Sensio\FrameworkBundle`. C'est un *bundle* activé par défaut sous Symfony2. Vérifiez juste que vous ne l'avez pas désactivé.

Vous pouvez ensuite vous servir de *DoctrineParamConverter*. La façon la plus simple de s'en servir est la suivante.

### 1. Côté route

Votre route ne change pas, vous gardez un paramètre {id} à l'ancienne. Exemple (tiré de `src/sdz/BlogBundle/Resources/config/routing.yml`):

**Code : Autre**

```
src/Sdz/BlogBundle/Resources/config/routing.yml
sdzblog_voir:
 pattern: /voir/{id}
 defaults: { _controller: SdzBlogBundle:Blog:voir }
```

### 2. Côté contrôleur

C'est dans le contrôleur que tout se joue et que la magie opère. Il vous suffit de changer la déclaration de votre action (dans `src/Sdz/BlogBundle/Controller/BlogController.php`).

- Avant :

**Code : PHP**

```
<?php public function voirAction($id)
```

- Après :

**Code : PHP**

```
<?php
// En haut du fichier
use Sdz\BlogBundle\Entity\Article;

// Et dans la classe
public function voirAction(Article $article)
```

Voici le code complet du contrôleur :

**Code : PHP**

```
<?php
use Sdz\BlogBundle\Entity\Article;

// ...

public function voirAction(Article $article)
{
 // Récupérons le titre de l'article.
 $article->getTitre();

 return $this->render('SdzBlogBundle:Blog:voir.html.twig', array('article' => $article));
}
```

Et voilà ! Vous pouvez maintenant utiliser directement `<?php $article->getTitre()` ou tout autre code utilisant `<?php $article` dans votre contrôleur. Pratique, n'est-ce pas ?

Voilà donc encore une astuce qui va vous permettre d'économiser des lignes de code ! Symfony2 nous aide vraiment : fini les lignes redondantes juste pour vérifier qu'un article existe. 😊

## Personnaliser les pages d'erreur

Avec Symfony2, lorsqu'une exception est déclenchée, le noyau l'attrape. Cela lui permet ensuite d'effectuer l'action adéquate.

Le comportement par défaut du noyau consiste à appeler un contrôleur particulier intégré à Symfony2 :

`TwigBundle:Exception:show`. Ce contrôleur récupère les informations de l'exception, choisit le *template* adéquat (un *template* différent par type d'erreur), passe les informations au *template* et envoie la réponse générée par ce *template*.

À partir de là, il est facile de personnaliser ce comportement : **TwigBundle** étant un... *bundle*, on peut le modifier pour l'adapter à nos besoins ! Mais ce n'est pas le comportement que nous voulons changer, c'est juste l'apparence de nos pages d'erreur. Il suffit donc de créer nos propres *templates* et de dire à Symfony2 d'utiliser nos *templates* et non ceux par défaut.

### Théorie : remplacer les templates d'un bundle

Il est très simple de remplacer les *templates* d'un *bundle* quelconque par les nôtres. Il suffit de créer le répertoire `app/Resources/NomDuBundle/views/` et d'y placer nos *templates* !

Nos *templates* doivent porter exactement les mêmes noms que ceux qu'ils doivent remplacer. Ainsi, si notre *bundle* utilise un *template* situé dans « ...`(namespace)/RépertoireDuBundle/Resources/views>Hello/salut.html.twig` », alors nous devons créer le *template* « `app/Resources/NomDuBundle/views>Hello/salut.html.twig` ».



Attention, le NomDuBundle en bleu correspond bien au nom du bundle, à savoir au nom du fichier que vous pouvez trouver à sa racine. Par exemple : `SdzBlogBundle` est le nom du bundle, mais il se trouve dans `(src)/Sdz/BlogBundle`.

Symfony2, pour chaque *template* qu'il charge, regarde d'abord dans le répertoire `app/Resources/` s'il trouve le *template* correspondant. S'il ne le trouve pas, il va ensuite voir dans le répertoire du *bundle*.

### Pratique : remplacer les templates Exception de TwigBundle

Maintenant qu'on sait le faire, il ne reste plus qu'à le faire. 

Créez donc le répertoire `app/Resources/TwigBundle/views/`.

En l'occurrence, les *templates* des messages d'erreur se trouvent dans le répertoire `Exception`, créons donc le répertoire `app/Resources/TwigBundle/views/Exception`.

Et au sein de ce répertoire, le *bundle* utilise la convention suivante pour chaque nom de *template* :

- il vérifie d'abord l'existence du `template error[code_erreur].html.twig`, par exemple, `error404.html.twig` dans le cas d'une page introuvable (erreur 404);
- si ce *template* n'existe pas, il vérifie l'existence du `template error.html.twig`, une sorte de page d'erreur générique.

Vous pouvez créer `error404.html.twig` pour les pages non trouvées et `error500.html.twig` pour les erreurs internes, ce sont deux des plus utilisées. Mais n'oubliez pas de créer `error.html.twig` également, sinon, vous aurez des pages d'erreur dépareillées en cas d'erreur (401, par exemple, pour un accès refusé).

### Le contenu d'une page d'erreur

Pour savoir quoi mettre dans ces *templates*, je vous propose de jeter un œil à celui qui existe déjà, `error.html` (il se trouve dans `vendor/symfony/src/Symfony/Bundle/TwigBundle/Resources/views/Exception`):

Code : HTML

```
<!DOCTYPE html>
<html>
 <head>
 <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
 <title>An Error Occurred: {{ status_text }}</title>
 </head>
 <body>
 <h1>Oops! An Error Occurred</h1>
 <h2>The server returned a "{{ status_code }} {{ status_text
}}".</h2>

 <div>
 Something is broken. Please e-mail us at [email] and let
 us know
 what you were doing when this error occurred. We will
 fix it as soon
 as possible. Sorry for any inconvenience caused.
 </div>
 </body>
</html>
```

Vous pouvez voir les différentes variables que vous pouvez utiliser : `{{ status_text }}` et `{{ status_code }}`. Fort de ça, vous pouvez créer la page d'erreur que vous souhaitez : vous avez toutes les clés.



Soyons d'accord : cette page d'erreur que l'on vient de personnaliser, c'est la page d'erreur générée en mode « prod » !



Remplacer la page d'erreur du mode « dev » n'a pas beaucoup d'intérêt : vous seul la voyez, et elle est déjà très complète. Cependant, si vous souhaitez quand même la modifier, alors cela n'est pas le `template error.html.twig` qu'il faut créer mais le `template exception.html.twig`. Celui-ci se trouve aussi dans le répertoire `Exception/`.

Retenez deux astuces en une :

- modifier les *templates* d'un *bundle* quelconque est très pratique, votre site garde ainsi une cohérence dans son design, et ce, que ce soit sur votre *bundle* à vous comme sur les autres ;
- personnaliser les pages d'erreur, ça n'est pas la priorité lorsque l'on démarre un projet Symfony2, mais c'est impératif avant de l'ouvrir à nos visiteurs.

## Utiliser la console directement depuis le navigateur !

La console est un outil bien pratique de Symfony2. Mais des fois, devoir ouvrir le terminal de Linux ou l'invite de commandes de Windows n'est pas très agréable. Et je ne parle pas des hébergements mutualisés qui n'offrent pas d'accès SSH pour utiliser la console !

Comment continuer d'utiliser la console dans ces conditions ? Ce chapitre est là pour vous expliquer cela !

### Théorie : Le composant Console de Symfony2

#### Les commandes sont en PHP

Nous l'avons déjà évoqué au cours de ce tutoriel, les commandes Symfony2 sont bien de simples codes PHP ! Effectivement on les exécute depuis une console, mais cela ne les empêche en rien d'être en PHP.

Et comme elles sont en PHP... elle peuvent tout à fait être exécutées depuis un autre script PHP. C'est en fait ce qui est déjà fait par le script PHP de la console, celui que l'on exécute à chaque fois : le fichier `app/console`. Voici son contenu :

**Code : PHP**

```
#!/usr/bin/env php
<?php

require_once __DIR__.'/bootstrap.php.cache';
require_once __DIR__.'/AppKernel.php';

use Symfony\Bundle\FrameworkBundle\Console\Application;
use Symfony\Component\Console\Input\ArgvInput;

$input = new ArgvInput();
$env = $input->getParameterOption(array('--env', '-e'),
getenv('SYMFONY_ENV') ?: 'dev');
$debug = !$input->hasParameterOption(array('--no-debug', ''));

$kernel = new AppKernel($env, $debug);
$application = new Application($kernel);
$application->run();
```

Comme vous pouvez le voir, ce fichier ressemble beaucoup au contrôleur frontal, `app.php`. Il charge également le Kernel. La seule chose qu'il fait de différent, c'est d'utiliser le **composant Console de Symfony2**, en instanciant la classe `Application` (ligne 15). C'est cet objet qui va ensuite exécuter les différentes commandes définies en PHP dans les *bundles*.

### Exemple d'une commande

Chaque commande est définie dans une classe PHP distincte, que l'on place dans le répertoire `Command` des *bundles*. Ces classes comprennent entre autres deux méthodes :

- `configure()` : Elle définit le nom, les arguments et la description de la commande ;
- `execute()` : Elle exécute la commande à proprement parler.

Prenons l'exemple de la commande `list`, qui liste toutes les commandes disponibles dans l'application. Elle est définie dans le fichier `vendor/symfony/src/Component/Console/Command>ListCommand.php`, dont voici le contenu :

**Code : PHP**

```
<?php

/*
 * This file is part of the Symfony package.
 *
 * (c) Fabien Potencier <fabien@symfony.com>
 *
 * For the full copyright and license information, please view the
 LICENSE
 * file that was distributed with this source code.
 */

namespace Symfony\Component\Console\Command;

use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Output\Output;
use Symfony\Component\Console\Command\Command;

/**
 * ListCommand displays the list of all available commands for the
 * application.
 *
 * @author Fabien Potencier <fabien@symfony.com>
 */
class ListCommand extends Command
{
```

```

 /**
 * {@inheritDoc}
 */
protected function configure()
{
 $this
 ->setDefinition(array(
 new InputArgument('namespace',
 InputArgument::OPTIONAL, 'The namespace name'),
 new InputOption('xml', null,
 InputOption::VALUE_NONE, 'To output help as XML'),
))
 ->setName('list')
 ->setDescription('Lists commands')
 ->setHelp(<<<EOF
The <info>list</info> command lists all commands:

<info>php app/console list</info>

You can also display the commands for a specific namespace:
<info>php app/console list test</info>

You can also output the information as XML by using the <comment>--xml</comment> option:
<info>php app/console list --xml</info>
EOF
);
}

/**
 * {@inheritDoc}
 */
protected function execute(InputInterface $input,
OutputInterface $output)
{
 if ($input->getOption('xml')) {
 $output->writeln($this->getApplication()->asXml($input-
>getArgument('namespace')), OutputInterface::OUTPUT_RAW);
 } else {
 $output->writeln($this->getApplication()->asText($input-
>getArgument('namespace')));
 }
}
}

```

Vous distinguez bien ici les deux méthodes qui composent la commande list.

En vous basant sur cet exemple, vous êtes d'ailleurs capable d'écrire votre propre commande : ce n'est vraiment pas compliqué !

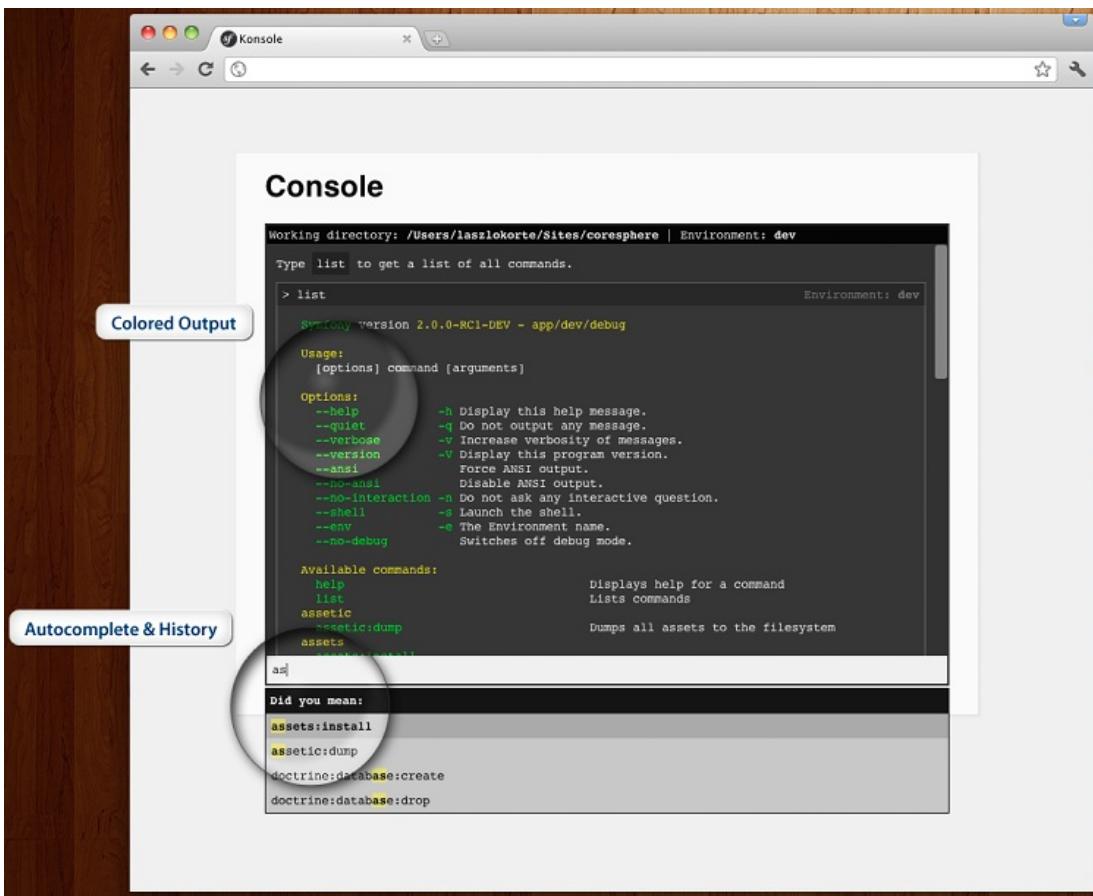
Mais revenons au but de ce chapitre, qui est de pouvoir utiliser ces commandes depuis le navigateur.

### Pratique : Utiliser un ConsoleBundle

#### ConsoleBundle ?

Vous le savez sûrement, la communauté de Symfony2 est très active, et un nombre impressionnant de *bundles* ont vu le jour depuis la sortie de Symfony2. Vous pouvez les retrouver presque tous sur ce site : <http://knxbundles.com/> qui les recense.

Il doit sûrement y avoir plusieurs *bundles* qui fournissent une console dans le navigateur, mais je vous propose d'en installer un en particulier : `CoreSphereConsoleBundle`. C'est un *bundle* simple qui remplit parfaitement sa tâche, et dont l'interface est très pratique. Voyez par vous-mêmes :



## Installer CoreSphereConsoleBundle

L'installation d'un tel *bundle* est vraiment simple, attaquons-la dès maintenant.

### Télécharger CoreSphereConsoleBundle

Pour installer ce *bundle*, je vous propose de télécharger une version que j'ai modifiée. J'ai entre autres résolu quelques petits bugs et traduit les messages en français. L'adresse du bundle est donc la suivante : <https://github.com/winzou/ConsoleBundle>.

Pour l'installer avec Composer, rajoutez la ligne suivante dans vos dépendances :

#### Code : JavaScript

```
// composer.json
"require": {
 // ...
 "winzou/console-bundle": "dev-master"
}
```

Puis mettez à jour vos dépendances grâce à la commande :

#### Code : Console

```
php ../composer.phar update
```

Si jamais vous n'avez pas Composer, ce que je déconseille, vous pouvez le télécharger à la main depuis Github. Cliquez sur *Download* et téléchargez l'archive (le format zip est recommandé pour les utilisateurs de Windows). Décompressez ensuite l'archive dans le répertoire `vendor/bundles/CoreSphere/ConsoleBundle`. Ensuite, enregistrez le namespace dans votre autoload comme ceci :

#### Code : PHP

```
<?php
// app/autoload.php
// ...

$loader->add('CoreSphere', __DIR__.'../../../vendor/bundles');

return $loader;
```

### Enregistrement du bundle dans le Kernel

Puis il faut enregistrer le bundle "CoreSphereConsoleBundle" dans app/AppKernel.php

#### Code : PHP

```
<?php

use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
 public function registerBundles()
 {
 $bundles = array(
 new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
 new Symfony\Bundle\SecurityBundle\SecurityBundle(),
 new Symfony\Bundle\TwigBundle\TwigBundle(),
 new Symfony\Bundle\MonologBundle\MonologBundle(),
 new
 Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
 new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
 new Symfony\Bundle\AsseticBundle\AsseticBundle(),
 new
 Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
 new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),

 // D'autres bundles que vous auriez déjà pu ajouter
);
 if (in_array($this->getEnvironment(), array('dev', 'test')))
 {
 $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
 $bundles[] = new
 Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
 $bundles[] = new
 Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
 $bundles[] = new
 Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();

 // D'autres bundles que vous auriez déjà pu ajouter

 // On enregistre ce bundle uniquement pour
 // l'environnement de développement évidemment
 $bundles[] = new CoreSphere\ConsoleBundle\CoreSphereConsoleBundle();
 }

 return $bundles;
 }
 // ...
}
```

### Enregistrement des routes

Pour paramétrer un bundle, on fait comme toujours : on lit sa documentation. La documentation se trouve soit dans le readme, soit dans le répertoire Resources/doc, cela dépend des bundles. Dans notre cas, elle se trouve dans le readme : [https://github.com/winzou/ConsoleBundl \[...\] ter/readme.md](https://github.com/winzou/ConsoleBundl [...] ter/readme.md)

Pour les routes, il faut donc enregistrer le fichier dans notre routing\_dev.yml. On ne les met pas dans routing.yml car la console ne doit être accessible qu'en mode dev, on a enregistré le bundle que pour ce mode. Ajoutez donc à la fin de app/config/routing\_dev.yml :

#### Code : Autre

```
console:
 resource: "@CoreSphereConsoleBundle/Resources/config/routing.yml"
```

### Publier les assets

L'installation touche à sa fin, il ne reste plus qu'à rendre disponible les fichiers JS et CSS du bundle, qui se fait comme vous le savez grâce à la commande suivante :

#### Code : Console

```
php app/console assets:install --symlink web
```

C'est fini ! Il ne reste plus qu'à utiliser notre nouvelle console.

## Utilisation de la console dans son navigateur

Par défaut, le *bundle* définit la route `/console` pour afficher la console. Allez donc à l'adresse [http://localhost/Symfony/web/app\\_dev.php/console](http://localhost/Symfony/web/app_dev.php/console) et profitez !



Bien entendu, pour exécuter des commandes Symfony2 depuis cette interface, il ne faut pas faire `php app/console la commande`, mais uniquement `la commande` ! Le script PHP du *bundle* n'utilise pas le script `app/console`, il utilise directement le composant Console.

Pour les utilisateurs de Windows, vous pouvez remarquer que le résultats des commandes est en couleur. Et oui, Symfony2 est plus fort que l'invite de commande de Windows, il gère les couleurs !

En plus de l'adresse `/console` dédiée, j'ai rajouté un petit bouton console, regardez en bas à droite dans la barre d'outils de Symfony :



Cliquez dessus, et une petite console s'ouvre par dessus votre page. Pratique pour exécuter une commande rapidement ! Pour enlever la console, recliquez sur le bouton.

## Prêt pour l'hébergement mutualisé

Et voilà vous êtes donc prêt pour utiliser la console de votre application sur les hébergements mutualisés qui n'offrent généralement pas d'accès SSH.

Vous disposez maintenant d'une console accessible depuis votre navigateur : cela va vous simplifier la vie croyez-moi ! 😊

N'hésitez pas à faire vos retours sur le *bundle* directement via les *issues* sur github : <https://github.com/winzou/ConsoleBundle/issues>.

## Déployer son site Symfony2 en production

Votre site est fonctionnel ? Il marche parfaitement en local, et vous voulez que le monde entier en profite ? Vous êtes au bon endroit, on va voir dans ce chapitre les points à vérifier pour déployer votre site sur un serveur distant.

### Préparer son application en local

Bien évidemment, la première chose à faire avant d'envoyer son application sur un serveur, c'est de bien vérifier que tout fonctionne chez soi ! Vous êtes habitué à travailler dans l'environnement de développement et c'est normal, mais pour bien préparer le passage en production, on va maintenant utiliser le mode production.

#### Vider le cache, tout le cache

Tout d'abord, pour être sûr de tester ce qui est codé, il faut vider le cache. Faites donc un petit :

**Code : Console**

```
php app/console cache:clear
```

Voici qui vient de vider le cache... de l'environnement de développement ! Et oui, n'oubliez donc jamais de bien vider le cache de production, via la commande :

**Code : Console**

```
php app/console cache:clear --env=prod
```

#### Tester l'environnement de production

Pour tester que tout fonctionne correctement en production, il faut utiliser le contrôleur frontal `app.php` comme vous le savez, et non `app_dev.php`. Mais cet environnement n'est pas très pratique pour détecter et résoudre les erreurs, vu qu'il ne les affiche pas du tout. Pour cela, ouvrez le fichier `web/app.php` on va activer le mode `debugger` pour cet environnement. Il correspond au deuxième argument du constructeur du Kernel :

**Code : PHP**

```
<?php
// web/app.php

// ...

$kernel = new AppKernel('prod', true); // Définissez ce 2e argument
// à true
```

Dans cette configuration, vous êtes toujours dans l'environnement de production, avec tous les paramètres qui vont bien : rappelez-vous certains fichiers comme `config.yml` ou `config_dev.yml` sont chargés différemment selon l'environnement. L'activation du mode `debugger` ne change rien à cela, mais permet d'afficher à l'écran les erreurs.



Pensez à bien remettre ce paramètre à `false` lorsque vous avez fini vos tests !



Lorsque le mode debugger est désactivé, les erreurs ne sont certes pas affichées à l'écran, mais elles sont heureusement répertoriées dans un fichier : `app/logs/prod`. Si l'un de vos visiteurs vous rapporte une erreur, c'est dans ce fichier qu'il faut aller regarder pour avoir le détail, les informations nécessaires à la résolution de l'erreur.

#### Soigner ses pages d'erreurs

En tant que développeur vous avez la chance de pouvoir utiliser l'environnement de développement et d'avoir de très jolies pages d'erreur, grâce à Symfony2. Mais mettez-vous à la place de vos visiteurs : créez volontairement une erreur sur l'une de vos pages (une fonction Twig mal orthographiée par exemple), et regardez le résultat depuis l'environnement de production (et sans le mode `debugger` bien sûr !). Voici ce que cela donne :

#### **Oops! An Error Occurred**

**The server returned a "500 Internal Server Error".**

Something is broken. Please e-mail us at [email] and let us know what you were doing when this error occurred. We will fix it as soon as possible. Sorry for any inconvenience caused.

Moche, n'est-ce pas ? C'est pour cela qu'il faut impérativement que vous personnalisiez les pages d'erreur de l'environnement de production. Un chapitre entier est dédié à ce point important, je vous invite à lire [Personnaliser les pages d'erreur](#).

#### Installer une console sur navigateur

En fonction de l'hébergement que vous avez, vous n'avez pas forcément l'accès SSH nécessaire pour exécuter les commandes Symfony2. Heureusement, les commandes Symfony2 sont de simples scripts PHP, il est alors tout à fait possible de les exécuter depuis un navigateur. Il existe des bundles qui émulent une console dans un navigateur, décrits dans un chapitre dédié : je vous

invite à lire le chapitre [Utiliser la console directement depuis le navigateur](#).

## Vérifier et préparer le serveur de production

### Vérifier la compatibilité du serveur

Evidemment, pour déployer une application Symfony2 sur votre serveur, encore faut-il que celui-ci soit compatible avec les besoins de Symfony2 ! Pour vérifier cela, on peut distinguer deux cas.

#### *Vous avez déjà un hébergeur*

Ce cas est le plus simple, car vous avez accès au serveur. Vous le savez, Symfony2 intègre un petit fichier PHP qui fait toutes les vérifications de compatibilité nécessaires, utilisons-le ! Il s'agit du fichier `web/config.php`, mais avant de l'envoyer sur le serveur il nous faut le modifier un petit peu. En effet, ouvrez-le, vous pouvez voir qu'il y a une condition sur l'IP qui appelle le fichier :

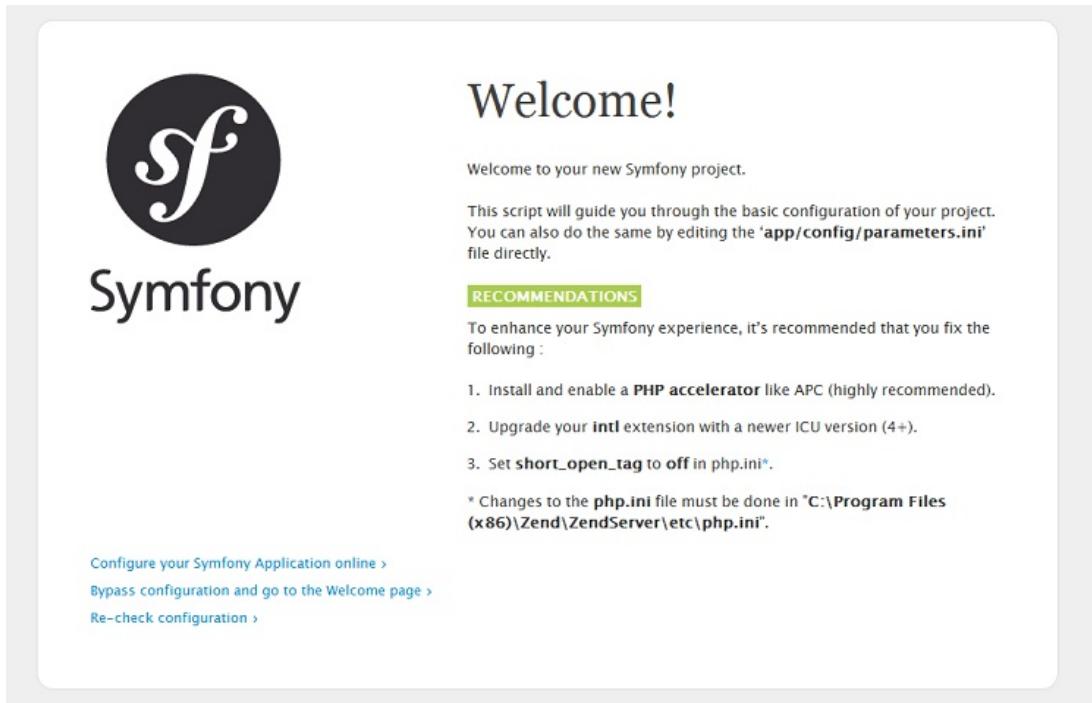
#### Code : PHP

```
<?php
// web/config.php

// ...

if (!in_array(@$_SERVER['REMOTE_ADDR'], array(
 '127.0.0.1',
 '::1',
))) {
 header('HTTP/1.0 403 Forbidden');
 exit('This script is only accessible from localhost.');
}
```

Comme ce fichier n'est pas destiné à rester sur votre serveur, supprimez simplement ce bloc et envoyez le fichier sur votre serveur. Ouvrez la page web qui lui correspond, par exemple [www.votre-serveur.com/config.php](http://www.votre-serveur.com/config.php) :



Comme vous le voyez, mon serveur est compatible avec Symfony2 car il n'y a pas de partie "Major Problems", juste des "Recommendations". Bien évidemment, essayez de résoudre les recommandations avec votre hébergeur/administrateur si cela est possible. Notamment, comme Symfony2 l'indique, installer un accélérateur PHP comme APC est très important, cela augmentera très très sensiblement les performances. Si elles n'étaient pas importantes en local, elles le seront en ligne !



Si vous avez envoyé seulement le fichier config.php, vous aurez bien sûr les deux problèmes majeurs comme quoi Symfony2 ne peut pas écrire dans les répertoires app/cache et app/logs. Pas d'inquiétude, on enverra tous les autres fichiers un peu plus tard.

#### *Vous n'avez pas encore d'hébergeur et en cherchez un compatible*

Dans ce cas, vous ne pouvez pas exécuter le petit script de test inclus dans Symfony2. Ce n'est pas bien grave, vous allez le faire à la main ! Voici les points obligatoires qu'il faut que votre serveur respecte pour pouvoir faire tourner Symfony2 :

- La version de PHP doit être supérieure ou égale à PHP 5.3.2 ;
- L'extension Sqlite3 doit être activée ;

- L'extension JSON doit être activée ;
- L'extension ctype doit être activée ;
- Le paramètre date.timezone doit être défini dans le php.ini.

Il y a bien entendu d'autres points qu'il vaut mieux vérifier, bien qu'ils ne soient pas obligatoires. La liste complète est disponible dans [la documentation officielle](#).

## Modifier les paramètres OVH pour être compatible

Certains hébergeurs permettent la modification de certains paramètres via les `.htaccess` ou l'interface d'administration. Il m'est bien sûr impossible de lister toutes les solutions pour chaque hébergements. C'est pourquoi ce paragraphe est uniquement à destination des personnes hébergées sur OVH, il y en a beaucoup et c'est un cas un peu particulier.

Vous savez que le PHP par défaut d'OVH est une branche de la version 4, or Symfony2 a besoin de la version 5.3.2 minimum. Pour cela, créez un fichier `.htaccess` à la racine de votre hébergement, dans le répertoire `www` :

**Code : Autre**

```
SetEnv SHORT_OPEN_TAGS 0
SetEnv REGISTER_GLOBALS 0
SetEnv MAGIC_QUOTES 0
SetEnv SESSION_AUTOSTART 0
SetEnv ZEND_OPTIMIZER 1
SetEnv PHP_VER 5_3
```

Ceci permettra notamment d'activer la version 5.3 de PHP, mais également de définir quelques autres valeurs utiles au bon fonctionnement de Symfony2.

## Déployer votre application

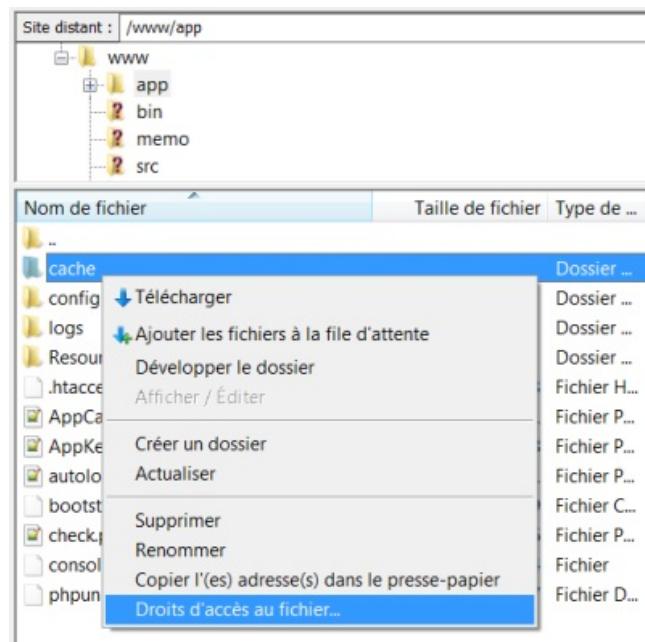
### Envoyer les fichiers sur le serveur

Dans un premier temps, il faut bien évidemment envoyer les fichiers sur le serveur. Pour éviter d'envoyer des fichiers inutiles et lourds, videz dans un premier temps le cache de votre application : celui-ci est de l'ordre des 1-10 Mo. Ensuite, envoyez tous vos fichiers et dossiers à la racine de votre hébergement, dans `www` / sur OVH par exemple.

## Régler les droits sur les dossiers app/cache et app/logs

Vous le savez maintenant, Symfony2 a besoin de pouvoir écrire dans deux répertoires : `app/cache` pour y mettre le cache de l'application et ainsi améliorer les performances, et `app/logs` pour y mettre l'historique des informations et erreurs rencontrées lors de l'exécution des pages.

Normalement, votre client FTP devrait vous permettre de régler les droits sur les dossiers. Avec FileZilla par exemple, un clic droit sur les dossiers `cache` et `logs` vous permet de définir les droits :



Assurez-vous d'accorder tous les droits (777) pour que Symfony2 puisse écrire à souhait dans ces dossiers.

## S'autoriser l'environnement de développement

Pour exécuter les commandes Symfony2, notamment celles pour créer la base de données, il nous faut avoir accès à l'environnement de développement. Or, essayez d'accéder à votre `app_dev.php`... accès interdit ! En effet, si vous l'ouvrez, vous remarquez qu'il y a le même test sur l'IP qu'on avait rencontré dans `config.php`. Cette fois-ci, ne supprimez pas la

condition car vous aurez besoin d'accéder à l'environnement de développement dans le futur. Il faut donc que vous complétiez la condition avec votre adresse IP. Obtenez votre ip sur [www.whatismyip.com](http://www.whatismyip.com), et rajoutez la :

#### Code : PHP

```
<?php
// web/app_dev.php

// ...

if (!in_array(@$_SERVER['REMOTE_ADDR'], array(
 '127.0.0.1',
 '::1',
 '123.456.789.1'
))) {
 header('HTTP/1.0 403 Forbidden');
 exit('You are not allowed to access this file. Check
 .basename(__FILE__).' for more information.');
}
```

Voilà vous avez maintenant accès à l'environnement de développement et, surtout, à la console 😊

## Mettre en place la base de données

Il ne manque pas grand chose avant que votre site ne soit opérationnel. Il faut notamment s'attaquer à la base de données. Pour cela, modifiez le fichier app/config/parameters.ini de votre serveur afin d'adapter les valeurs des paramètres database\_\*.

Généralement sur un hébergement mutualisé vous n'avez pas le choix dans la base de données, et vous n'avez pas les droits pour en créer. Mais si ce n'est pas le cas, alors il faut créer la base de données que vous avez renseigné dans le fichier parameters.ini, en exécutant cette commande :

#### Code : Console

```
php app/console doctrine:database:create
```

Puis, dans tous les cas, remplissez la base de données avec les tables correspondantes à vos entités :

#### Code : Console

```
php app/console doctrine:schema:update --force
```

## S'assurer que tout fonctionne

Ca y est, votre site devrait être opérationnel dès maintenant ! Vérifiez que tout fonctionne bien dans l'environnement de production.



En cas de page blanche ou d'erreur 500 pas très bavarde : Soit vous allez voir les logs dans app/logs/prod, soit vous activez le mode debugger dans app.php comme on l'a fait précédemment. Dans tous les cas pas de panique : si votre site fonctionnait très bien en local, l'erreur est souvent très bête sur le serveur (problème de casse, oubli, etc.).

## Avoir de belles URL

Si votre site fonctionne bien, vous devez sûrement avoir ce genre d'URL pour l'instant : [www.votre-site.com/web/app.php](http://www.votre-site.com/web/app.php). On est d'accord, on ne va pas rester avec ces horribles URL !

Pour cela il faut utiliser l'[URL Rewriting](#), une fonctionnalité du serveur web Apache (rien à voir avec Symfony2). L'objectif est que les requêtes "/blog" et "/css/style.css" arrivent sur "/web/blog" et "/web/css/style.css" respectivement. Rajoutez donc ces lignes dans un .htaccess à la racine de votre serveur :

#### Code : Autre

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

C'est tout ! En effet, c'est déjà bon pour les fichiers CSS, mais pour l'URL "/blog" il faut qu'au final elle arrive sur "/web/app.php/blog". En fait il y a déjà un .htaccess dans le répertoire /web, ouvrez-le, il contient ce qu'il faut. Pour résumer, l'URL "/blog" va être réécrite en "/web/blog" par notre .htaccess à la racine, puis être à nouveau réécrite en "/web/app.php/blog"

par le `.htaccess` de Symfony2 situé dans le répertoire /web.

## Et profitez !

Et voilà votre site est pleinement opérationnel, profitez-en !



Et n'oubliez pas, à chaque modification de code source que vous envoyez sur le serveur, vous devez obligatoirement vider le cache de l'environnement de production ! L'environnement de production ne fonctionne pas comme l'environnement de développement : il ne vide jamais le cache tout seul, jamais !

La liste des astuces n'est pas définitive et ne le sera jamais : j'en rajouterais dès que je les rencontrerai moi-même lorsque je coderai. Si vous connaissez une astuce intéressante qui n'est pas encore traitée dans cette partie, vous pouvez aussi m'envoyer un MP pour que je puisse l'ajouter ici. 😊

## Avancement du cours

Ce cours est en pleine phase de rédaction. Je publierai les parties au fur et à mesure afin qu'un maximum de personnes puisse profiter des premiers cours francophones sur **Symfony2**. N'hésitez pas à repasser plus tard ou à vous abonner au flux RSS du **cours**. 😊

## En attendant

En attendant les prochains chapitres... ne chômez pas ! La [documentation officielle](#) est très bien faite, n'hésitez pas à parcourir les [nombreux bundles](#) qui existent déjà, mais surtout, entraînez-vous !

Vous pouvez également visiter [mon blog sur Symfony2](#), dans lequel je vous tiens au courant de l'avancée du tutoriel ainsi que des trucs et astuces très pratiques sur Symfony2.

## Licences

Certaines images de ce tutoriel sont tirées de la documentation officielle. Elles sont donc soumises à la licence suivante :

**Citation : Sensio Labs**

Copyright (c) 2004-2010 Fabien Potencier

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.