

- [3 291 Zéros connectés](#)
- [272 463 Zéros inscrits](#)
- [Inscription](#)
- [Connexion](#)

- o Pseudo
- o
- o Mot de passe
- o
- o [Mot de passe oublié ?](#)
- o ☐ Connexion auto.
- o

Symfony2 - Un tutoriel pour débiter 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.

Informations sur le tutoriel

Auteur :

- [winzou](#)

Difficulté :

Temps d'étude estimé : 21 jours

Télécharger en PDF: Seuls les cours publiés sous licence Creative Commons par leur auteur peuvent être téléchargés en PDF.

Historique des mises à jour

[Rester informé grâce au flux RSS](#)

- Le 06/12/2011 à 18:52:31
Correction d'un code
- Le 18/11/2011 à 18:00:12
Quelques coquilles sur le chapitre Doctrine2
- Le 16/11/2011 à 20:58:54
Correction d'une coquille sur Doctrine2

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. 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, il est inutile de poursuivre la lecture de ce cours. Symfony2 requiert ces bases, et si vous ne les avez pas, vous risquez de perdre votre temps. C'est comme acheter un A380 sans savoir piloter : c'est joli mais ça ne sert à rien.

Partie 1 : Les bases de Symfony2 : « Hello World »

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

Symfony2, un framework PHP

Symfony2 est un *framework* PHP, c'est-à-dire un ensemble de fichiers PHP qui a pour but de vous simplifier la vie. Sympa, non ?

Dans cette partie, nous allons tout d'abord voir ce qu'est vraiment un *framework*, car Symfony2 est avant tout un *framework* comme un autre. Nous passerons ensuite à la pratique en le téléchargeant.

Les objectifs d'un framework

Un framework, qu'est-ce que c'est ?

Introduction

L'objectif de ce chapitre n'est pas de vous fournir toutes les clés pour concevoir un *framework*. Je vais juste vous exposer rapidement les intérêts, les pour et les contre de l'utilisation d'un tel outil.

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*, que ce soit en PHP, en JavaScript ou n'importe quel autre langage non Web.

Un *framework* est un script conçu par un ou plusieurs développeurs et 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.

Un framework en pratique

En pratique, un *framework* tel que Symfony2 n'est pas un outil évident à prendre en main ; néanmoins, il est très puissant. Justement, il n'est pas facile car il est puissant. En effet, pour s'adapter aux besoins de chaque développeur (et donc de chaque site Internet), un *framework* doit être très flexible, il doit pouvoir faire ce qu'on lui dit de faire. Mais pour être flexible, il doit imposer des règles... Paradoxal ? Non, obligatoire pour s'en sortir !

C'est donc un outil qui impose certaines règles au développeur final. Heureusement, ce sont généralement des règles qui permettent en même temps de bien organiser son code. Ainsi, le simple fait de « suivre les règles » vous empêche d'écrire un « mauvais code ».

Les avantages d'un framework

L'avantage le plus important, sans aucun doute, ce sont les mises à jour du *framework*. Imaginez que vous codiez vous-même tout ce qui est connexion utilisateur, session, moteur de template, etc. Il est impossible de coder sans *bug*, donc à chaque *bug* déclaré sur votre code, vous devrez le corriger vous-même : normal, c'est votre code. Imaginez maintenant que toutes ces parties, indispensables pour un site Web, mais qui finalement ne sont pas votre tasse de thé, soient fournies par quelqu'un d'autre. À chaque fois que vous ou les milliers d'autres utilisateurs trouverez un *bug*, les auteurs du *framework* s'occuperont de le corriger, et vous n'aurez qu'à suivre les mises à jour ! Vous pouvez enfin vous concentrer sur le cœur de votre site.

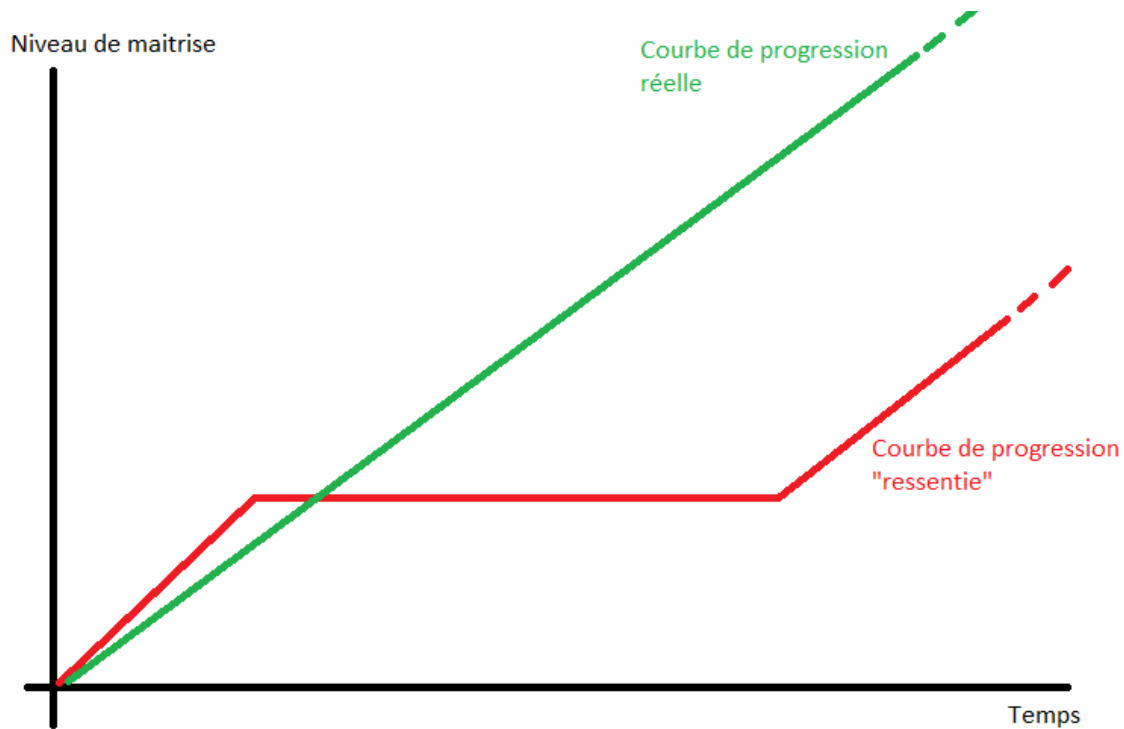
L'autre avantage d'utiliser un *framework* pour son site Internet, c'est lorsque d'autres développeurs vont vous rejoindre : s'ils connaissent déjà le *framework* en question, alors ils s'intégreront très rapidement à votre projet. Votre code vous est propre, mais comme vous suivez les règles du *framework*, toute personne connaissant ce *framework* pourra lire ce que vous aurez écrit avec une facilité déconcertante. C'est un avantage énorme si vous travaillez à plusieurs, que ce soit dès le début du projet ou lorsque votre projet prendra de l'ampleur.

La « barrière d'entrée » d'un framework

Forcément, pour proposer tous ces avantages, le *framework* a un défaut que l'on appelle la « barrière d'entrée ». C'est-à-dire... qu'il vous faut un cours comme celui-ci pour apprendre à le maîtriser. 🤪 Les règles, les conventions à respecter, les noms, les services à connaître, etc. Cela ne s'apprend pas en un jour. C'est un investissement à faire au début, mais qui vous sera bénéfique par la suite. 😊

En un seul mot : **ne vous découragez pas !**

Surtout quand vous atteindrez le fameux « palier ressenti ». Symfony2, c'est comme beaucoup de choses (les cours de conduite, etc.) : à un moment donné, vous avez l'impression de ne plus progresser. Il n'en est rien ! En réalité, vous n'arrêtez pas de progresser, vous consolidez simplement vos acquis. Regardez ce que mes talents de graphiste vous proposent pour illustrer cela :



Gardez en tête la courbe verte, la courbe rouge n'est qu'une représentation de votre esprit. 🤖

Framework VS PHP « simple »

Exemple d'une page d'un blog

Prenons un exemple, [celui de la documentation de Symfony2](#) :

Code : PHP - [Sélectionner](#)

```
<?php
// index.php

$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

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

<html>
<head>
<title>List of Posts</title>
</head>
<body>
<h1>List of Posts</h1>
<ul>
<?php while ($row = mysql_fetch_assoc($result)): ?>
<li>
<a href="/show.php?id=<?php echo $row['id'] ?>">
<?php echo $row['title'] ?>
</a>
</li>
<?php endwhile; ?>
</ul>
</body>
</html>

<?php
mysql_close($link);
```

Ce fichier par exemple, comme vous pouvez le voir, affiche les articles d'un blog. C'est très simple, très rapide et développé sans *framework*. Si vous lisez ce cours, vous avez certainement écrit ce genre de code, et c'est tout à votre honneur.

Mais essayons maintenant de l'analyser. Voici les quatre problèmes majeurs que l'on peut détecter rien que dans ces quelques lignes :

- **aucune gestion des erreurs** : si la connexion au serveur MySQL échoue, que fait ce code ? Il affiche une erreur la plus moche qui soit, du type :

Citation : PHP

Warning: mysql_connect() [function.mysql-connect]: php_network_getaddresses: gethostbyname failed. errno=0 in \backup\www\test.php on line 4

Vous souhaitez vraiment que vos visiteurs voient ça ?

- **aucune organisation** : dès que ce site grossira un peu, il deviendra très difficile à maintenir car tout est en vrac, au même endroit ;
- **impossible à réutiliser** : comme tout est dans le même fichier, il est impossible de réutiliser une partie du code. Imaginez que la liste des articles apparaisse sur une autre page, comment faire ? Un copier-coller ? Voyons, un peu de sérieux !
- **dépendance à MySQL** : dans le fichier, vous appelez des fonctions MySQL, mais si l'on devait changer de base de données, on va toutes les réécrire ?

Conclusion : pour « afficher une liste d'articles de blog », ce code est plus simple et plus rapide que son équivalent Symfony2, c'est clair. Mais pour « avoir un site flexible et pouvant être maintenu qui, entre autres, affiche une liste d'articles de blog », nous verrons tout au long de ce cours que Symfony2 va prendre indéniablement le dessus !

Architecture MVC

Architecture MVC... Vous avez certainement déjà entendu ce nom. Je ne vais pas rentrer dans les détails, mais voici les grandes lignes de cette fameuse architecture.

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** : son rôle est de générer la réponse à la requête HTTP demandée par notre visiteur. Le contrôleur contient la logique de notre site Internet et va se contenter « d'utiliser » les autres composants : le modèle et la vue. 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 `<?php if() ?>`, on s'en sort très bien ;
- **le modèle** : 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** : 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'appeler le modèle et la vue en leur attribuant des tâches précises.

Symfony2 respecte évidemment l'architecture MVC ! Alors, pourquoi ne pas profiter d'un exemple de contrôleur Symfony2 ? En voici un :

Code : PHP - Sélectionner

```
<?php

namespace FOS\UserBundle\Controller;

class UserController extends ContainerAware
{
    // Liste des utilisateurs
    public function listAction()
    {
        // On récupère la liste : ici, on fait appel au modèle « utilisateur ». En une
        // ligne, nous récupérons un tableau avec la liste des utilisateurs !
        $users = $this->get('fos_user.user_manager')->findUsers();

        // On affiche cette liste : ici, on fait appel à la vue, qui contient toutes les
        // balises <ul> et compagnie pour afficher la liste.
        return $this->render('FOSUserBundle:User:list.html.twig', array(
            'users' => $users
        ));
    }

    // Profil d'un utilisateur en particulier de pseudo $username
    public function showAction($username)
    {
        // On utilise le modèle « utilisateur ».
        $user = $this->findUserBy('username', $username);

        // On appelle la vue.
        return $this->render('FOSUserBundle:User:show.html.twig', array(
            'user' => $user
        ));
    }

    // Édition d'un utilisateur de pseudo $username
    public function editAction($username)
    {
        // On emploie le modèle « utilisateur ».
        $user = $this->findUserBy('username', $username);

        // On utilise le service (une sorte de modèle) « formulaire ».
        $form = $this->get('fos_user.form.user');
        $formHandler = $this->get('fos_user.form.handler.user');

        // On demande au modèle « formulaire » de valider le formulaire.
        if ($formHandler->process($user))
        {
            // On demande au modèle « utilisateur » de mettre à jour l'utilisateur.
        }
    }
}
```

```

        // On demande au modele « gestion des URL » l'URL du profil de
        l'utilisateur de pseudo $username.
        $userUrl = $this->generateUrl('fos_user_user_show', array('username' =>
        $user->getUsername()));

        // On fait une redirection vers cette URL.
        return new RedirectResponse($userUrl);
    }

    // Si le formulaire n'est pas valide ou n'a pas encore été rempli, on appelle la
    vue pour afficher le formulaire d'édition.
    return $this->render('FOSUserBundle:User:edit.html.twig', array(
        'form' => $form->createView(),
        'username' => $user->getUsername()
    ));
}
}

```

Comme vous pouvez le voir, le contrôleur est réduit à un code très simple : il n'y a rien de complexe, ici. Si vous ne comprenez pas le code, **c'est normal**, il fait appel à des fonctions que vous ne connaissez pas encore. Mais le nom des fonctions est plutôt clair, et le code dans l'ensemble ne paraît pas compliqué finalement.

En tout cas, retenir bien ces noms : contrôleur (*controller*), modèle (*model* ou *entity*) et vue (*view*) ; nous allons nous en servir en permanence avec Symfony2. Je vais vous guider dans la création de tous ces composants et vous pourrez vous rendre compte des bienfaits de cette architecture et de Symfony2 tout entier.

Pour aller plus loin dans la définition de l'architecture MVC, je vous propose de lire [le cours de vincent1870 sur le MVC](#). C'est un très bon approfondissement !

Un framework va bien au-delà !

Par rapport au PHP « simple », notre *framework* Symfony2 intègre l'architecture MVC, la gestion des erreurs, la « ré-utilisabilité » du code, l'abstraction de la base de données utilisée, etc. Vous pouvez dès maintenant voir à quel point Symfony2 va vous poser problème aujourd'hui ! Mais également comment il vous sauvera la vie demain. 😊

Qu'est-ce que Symfony2 ?

Un framework français

Eh oui, Symfony2, l'un des meilleurs *frameworks* PHP au monde (si ce n'est le meilleur), est bien un *framework* français ! Son auteur 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, d'Europe, des États-Unis, etc. C'est grâce au talent de Fabien et à la générosité de la communauté que Symfony2 a vu le jour.

Un framework très utilisé

symfony 1 est l'un des *frameworks* les plus utilisés au monde avec ZendFramework. Aujourd'hui, beaucoup d'entreprises dans le domaine de l'Internet (dont Simple IT !) recrutent des développeurs capables de développer sous ces deux *frameworks*. Ces développeurs pourront ainsi se greffer aux projets de l'entreprise très rapidement, car ils en connaîtront déjà les grandes lignes.

Symfony2 représente l'avenir de symfony. Bien que différent dans sa conception, il est plus rapide et plus souple que la première version. 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 découpé en composants

Symfony2 est bien conçu, ce qui permet d'éviter les dépendances entre les différents composants. Une dépendance, c'est lorsque vous modifiez une partie de votre code et que cela crée un *bug* dans une autre partie qui, *a priori*, n'avait vraiment rien à voir. 😞 Symfony2 n'est pas du tout comme ça et pour preuve : chacun de ses composants peut être utilisé tout seul dans un projet qui n'utilise pas Symfony2. Voici la liste des composants de Symfony2 : <http://symfony.com/components>.

Télécharger

Obtenir Symfony2

C'est très simple. Rendez-vous sur le site de Symfony2, rubrique « Download » : <http://symfony.com/download>. Téléchargez la version .zip ou .tgz, comme vous voulez, mais prenez-la avec *vendors* (pas la version *without vendors*, donc).

Une fois l'archive téléchargée, décompressez les fichiers dans votre répertoire **www** habituel. Pour la suite du tutoriel, je considérerai que vous les avez décompressés dans **www/Symfony**.

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>. 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 vous pouvez régler si ça vous amuse, mais qui sont facultatifs.



Un petit bug s'est glissé dans la version 2.0.0 de Symfony2. S'il vous demande de mettre à jour votre version d'APC alors que vous n'avez pas APC d'installé sur votre machine, vous pouvez ignorer ce message ! Symfony2 n'a pas besoin d'APC pour fonctionner. Ce message s'adresse à ceux qui ont déjà APC d'installé, mais sous une ancienne version.

Les répertoires de Symfony2

Rendez-vous dans le dossier où se trouve le *framework* Symfony2, vous trouverez plusieurs répertoires. Ces répertoires ne sont pas là par hasard, ils reflètent l'organisation de Symfony2. Voyons rapidement à quoi ils correspondent.

Répertoire	Rôle
app	Contient principalement la configuration de votre site dans son ensemble (pas votre code).
app/Resources	Contient les fichiers de langue, certaines vues, etc.
app/cache	Nous n'irons jamais dedans, il contient les fichiers de cache générés par Symfony2 pour accélérer l'exécution des pages.
app/config	Contient tous les fichiers de configuration.
app/log	Contient les fichiers de <i>log</i> des erreurs de votre site, peu utilisés pendant le développement car les erreurs s'afficheront à l'écran. Mais en mode « production » (ce que vos visiteurs verront), c'est le seul endroit où vous pourrez repérer les erreurs et leur description car elles ne s'afficheront pas à l'écran de l'internaute.
bin	Contient des scripts pour mettre à jour automatiquement les différents <i>bundles</i> et librairies utilisés par Symfony2.
src	Contient votre code ! C'est ici que nous passerons le plus clair de notre temps.
web	Contient tous les fichiers destinés à vos visiteurs : images, fichiers CSS et JavaScript, etc.

Concrètement, les deux répertoires que nous utiliserons le plus souvent seront `app/config/` pour gérer la configuration de notre site et `src/` pour notre code.

Exécuter

Exécuter Symfony2

Si vous vous rendez à l'adresse <http://localhost/Symfony>, malheureusement, vous n'aurez pas grand-chose. C'est un piège !

Le répertoire web

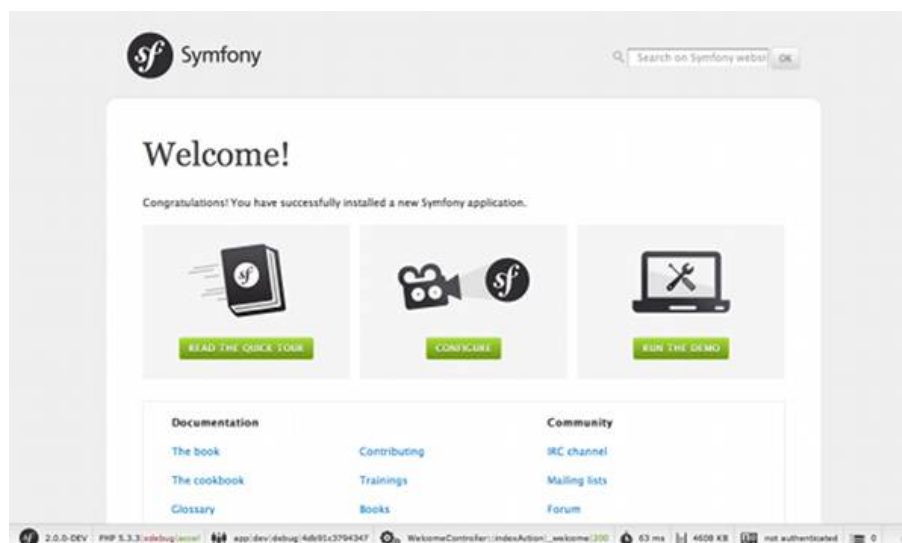
En réalité, ce qui est censé être disponible au public se trouve dans le répertoire `web`. C'est ici que se trouvera l'équivalent d'un fichier `index.php`, ainsi que toutes vos images et fichiers CSS, JavaScript, etc.

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.

Le « Front Controller »

Seulement, même dans le répertoire `web`, vous ne trouverez pas de fichier `index.php` 😞 Allez-y, vérifiez. En réalité, le fichier à exécuter est `app_dev.php`. Ce fichier s'appelle le « Front Controller », un nom bien compliqué juste pour dire que toutes les requêtes passent par lui.

Allez donc sur http://localhost/Symfony/web/app_dev.php et profitez !



Page d'accueil de Symfony2

En allant dans le répertoire `web`, vous avez sûrement remarqué qu'il existait aussi un fichier `app.php` tout court. En fait, pour nous simplifier la vie de développeur, Symfony2 dispose de deux modes :

deux « modes » :

- un mode « dev », lorsque vous exécutez `app_dev.php`. Ce mode permet d'afficher les erreurs, de donner plein d'informations en cas d'erreur, d'activer une *toolbar* pour obtenir plusieurs informations sur la page en cours, etc. C'est donc le mode que l'on utilisera pour programmer ;
- un mode « prod », lorsque vous exécutez `app.php`. Ce mode désactive l'affichage des erreurs ou de toute information utile au débogage. C'est le mode pour vos visiteurs, pour que ces derniers ne voient pas ce qu'il se passe dans les coulisses. En cas d'erreur, ils auront une jolie page d'erreur et non un message d'insulte en anglais comme quoi l'« Exception `OnCoreDeleteUnauthorizedTryFailed` has been thrown ». 🙄

Essayez-les ! Allez sur http://localhost/Symfony/web/app_dev.php et vous aurez une *toolbar* en bas de l'écran avec plein d'informations. 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. 🙄)

Pour voir le comportement 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\[...\]quinexistepas](http://localhost/Symfony/web/app_dev.php[...]quinexistepas). 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.

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. 🙄



Notez que nous n'avons pas encore touché à une seule ligne de code, et déjà nous avons découvert une fonctionnalité très pratique de Symfony2 : les modes « dev » et « prod ». 😊

Vous savez maintenant ce qu'est un *framework*, et en plus, nous avons réussi à exécuter Symfony2 : on s'amuse déjà beaucoup ! 😊 Mais concrètement, vous n'en savez pas encore beaucoup à son sujet, sur son fonctionnement, etc.

Je vous propose d'attaquer dès maintenant le prochain chapitre, théorique, sur les notions essentielles de Symfony2.

Vous avez dit Symfony2 ?

Dans ce chapitre, nous allons voir comment fonctionne Symfony2 à l'intérieur. Nous n'entrerons pas dans les détails, c'est bien trop compliqué, le but étant 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 ? 😊

Les requêtes HTTP

Cheminement d'une requête HTTP

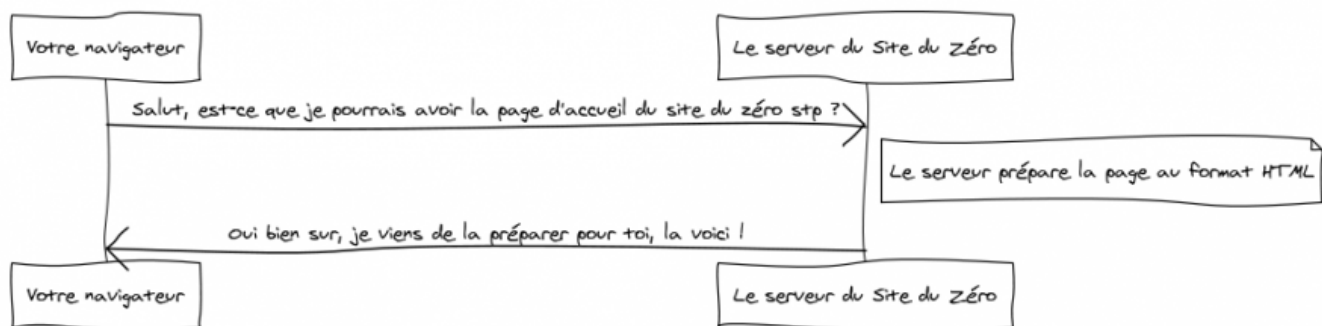
Pourquoi ?

Vous devez sûrement vous demander quel est le rapport avec Symfony2.

En fait, les développeurs de Symfony2 sont partis du principe même d'une requête HTTP pour construire le noyau. Vous retrouverez donc beaucoup de concepts HTTP en utilisant Symfony2, notamment les objets *Request* et *Response*.

Introduction

HTTP (*HyperText Transfer Protocol*) est un langage texte qui permet à deux machines de communiquer l'une avec l'autre. C'est tout ! Par exemple, lorsque vous accédez au Site du Zéro, voici ce qu'il se passe entre vous et le serveur de ce dernier :



www.websequencediagrams.com

1. Votre navigateur demande au serveur : « Salut, est-ce que je pourrais avoir la page d'accueil du siteduzero stp ? ».
2. Le serveur reçoit la requête et travaille pour construire la réponse.
3. Le serveur renvoie la réponse à votre navigateur qui n'a plus qu'à vous l'afficher.

Vous trouverez plus de détails sur les requêtes HTTP dans le chapitre correspondant du cours de M@teo21 : <http://www.siteduzero.com/tutoriel-3-197288-introduction-a-cha.html#chap-2>

La requête HTTP

La requête HTTP est ce que votre navigateur envoie. Voici un exemple de requête HTTP :

Code : Console - Sélectionner

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

Ce n'est rien d'autre que du texte ! Cette requête est simple, mais il peut en réalité y avoir plus de lignes pour préciser la langue, les cookies, les données de formulaire, etc. Le serveur ne vous connaît que par cette requête (il ne se souvient pas de vos requêtes précédentes), il faut donc que cette requête soit complète. Vous retrouverez toutes ces informations dans l'objet **Request** de Symfony2. Nous verrons bien sûr comment accéder à cet objet.

La réponse HTTP

La réponse HTTP est ce que le serveur vous renvoie. Par exemple :

Code : Console - Sélectionner

```
HTTP/1.1 200 OK
Date: Sat, 02 Apr 2011 21:05:05 GMT
Server: lighttpd/1.4.19
Content-Type: text/html

<html>
  <!-- Le code HTML du Site du Zéro. -->
</html>
```

La réponse contient donc :

- quelques lignes de *header* en haut pour donner des informations précises à votre navigateur : la langue, l'encodage, etc. ;
- le contenu HTML de la page renvoyée que le navigateur va vous afficher.

Le numéro (200 ici) après le « HTTP/1.1 » est le code réponse HTTP. La liste complète des codes possibles est disponible sur [Wikipédia](#). Vous pouvez confirmer que 200 correspond bien à « OK tout va bien », c'est le code que presque toutes vos pages vont renvoyer. Mais si vous tombez sur une page introuvable, vous devriez obtenir le code 404. Si vous tombez sur une page à laquelle vous n'avez pas accès (un forum privé par exemple), vous devriez avoir le code 401. Aucune inquiétude : Symfony2 et son objet **Response** sont là pour faire pas mal de travail à votre place et retourner les bons codes HTTP. 😊

Conclusion

Ce petit paragraphe n'est qu'une introduction aux requêtes HTTP. Il vous donne suffisamment d'informations pour bien démarrer avec Symfony2.



Retenez bien ceci : le serveur reçoit une **requête** et en fonction de cette requête, il doit construire et envoyer une **réponse**.

La vision de Symfony2 autour d'une requête HTTP

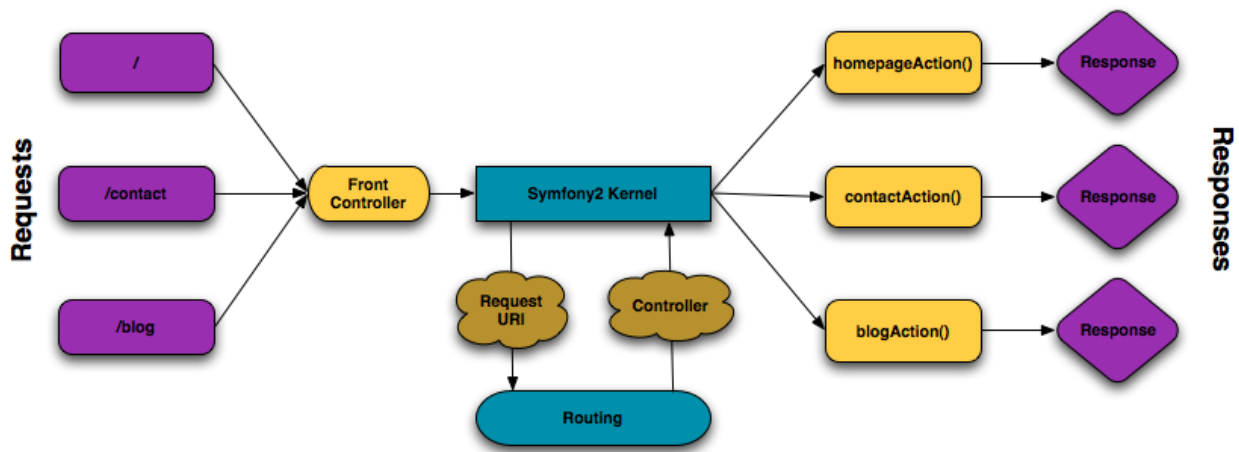
Symfony2 et les requêtes HTTP

Enfin, parlons un peu de Symfony2, ce pour quoi vous lisez ce cours. 😊

En fait, vous connaissez déjà beaucoup de choses sur Symfony2, je vous les ai distillées tout au long de ces premiers chapitres :

- Symfony2 est construit autour d'une requête HTTP et suit donc le processus Requête -> Calcul -> Réponse ;
- Symfony2 emploie l'architecture MVC et utilise donc contrôleurs, modèles et vues ;
- Symfony2 utilise en plus des services qui sont une sorte de modèle.

Fort de ces 3 points, voici comment Symfony2 traite une requête :



Un explication s'impose.

La requête (ou « request »)

La première rangée en violet représente les différentes requêtes que votre site va devoir gérer, par exemple : « GET / » ou « GET /contact » ou « GET /blog ». C'est la requête de base à partir de laquelle Symfony2 devra travailler.

Le contrôleur frontal (ou « front controller »)

Un nom un peu barbare juste pour représenter le fichier `app_dev.php`. Cela signifie simplement que c'est ce fichier qui reçoit toutes les requêtes et qui appelle le noyau (ou « kernel ») de Symfony2. En d'autres termes : toutes les requêtes passent par ce fichier, et ce fichier ne fait qu'appeler le noyau.

Le noyau (ou « kernel »)

Le contrôleur frontal appelle donc le noyau et lui dit : « Voilà, j'ai reçu une requête, traite-la s'il te plaît. » Le noyau, lui, c'est un peu le maître : il prend les choses en main.

- Première étape : il faut savoir quel contrôleur appeler. Pour cela, il le demande au service *routing* en lui donnant l'URL appelée.
- Le service *routing* fait son travail : il fait correspondre l'URL à l'une de ses nombreuses routes et renvoie le nom du contrôleur à appeler au noyau.
- Seconde étape : maintenant que nous savons quel contrôleur exécuter, exécutons-le. Le noyau dit donc au bon contrôleur : « Maintenant, c'est à toi de jouer. Tu dois absolument retourner une réponse à notre client, débrouille toi ! ».

Le contrôleur (ou « controller »)

La bonne méthode du contrôleur (dernière rangée jaune) est exécutée. Le contrôleur fait son travail et utilise tous les outils à sa disposition : services, modèles, vues, etc. Sa seule obligation est de retourner une réponse avec un objet de type *Response*.

La réponse (ou « response »)

La réponse retournée par le contrôleur est ainsi directement envoyée au visiteur, le contrôleur étant le dernier acteur du processus Symfony2.

Les points sur lesquels le développeur intervient

Le développeur qui utilise Symfony2 (nous !) n'intervient pas sur tous ces points, et heureusement. En fait, il ne fait plus grand-chose : c'est là toute la magie de Symfony2. 🧙 Les deux seuls points sur lesquels nous interviendrons seront le routeur et le contrôleur.

Le routeur (« router » ou « service routing »)

Le routeur fait la correspondance entre l'URL et le contrôleur. Par exemple, l'URL `/blog/list` correspond à la page qui affiche la liste des articles. Ça, Symfony2 ne peut pas le deviner ! C'est notre boulot de le lui indiquer et de toute façon, c'est mieux ainsi : je veux décider moi-même quelles URL adopter pour mon site. Nous ne toucherons pas au service *routing* en lui-même (c'est une classe PHP complexe), heureusement, nous définirons simplement les routes que le routeur devra utiliser. Les routes sont très faciles à définir : cela se fait directement dans un fichier texte. Chaque route fait correspondre une ou plusieurs URL à un seul contrôleur. Vous avez compris le principe ? Super ! Nous verrons les détails pratiques dans le prochain chapitre, ce n'est pas le sujet ici.

Le contrôleur (ou « controller »)

Le contrôleur, rappelez-vous, est la logique de notre site. Il est alors évident que nous devons l'écrire nous-mêmes ! C'est sur les contrôleurs et leurs modèles et vues que nous passerons le plus de temps.

Conclusion

La magie de Symfony2 onère ici : vous avez vu le processus de traitement d'une requête en entier ? Vous avez aussi vu les deux seuls points dont nous devons nous occuper ? C'est Symfony2

La magie de Symfony2 opère ici : vous avez vu le processus de traitement d'une requête en entier ; vous avez aussi vu les deux seuls points dont nous ne nous occupons pas. C'est Symfony2 qui fait tout le reste, c'est en ça qu'il nous fait gagner énormément de temps !

Symfony2 et ses bundles

La découpe en « bundles »

Le concept

Vous avez peut-être déjà entendu parler des *bundles* dans Symfony2. Pour faire simple, ce sont des « briques d'applications ». C'est évident, non ? 😊

Plus sérieusement, il s'agit vraiment de ça. Un *bundle* est un ensemble de fichiers qui contient tout ce qu'il faut pour remplir une fonctionnalité. Plein de *bundles* existent déjà, et parmi eux, nous pouvons citer :

- **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 commentaire à n'importe laquelle de vos pages !
- **GravatarBundle** : c'est un *bundle* destiné à gérer les avatars depuis le service web **Gravatar**. Concrètement, il fournit une extension à Twig pour pouvoir afficher un avatar issu de Gravatar via une simple fonction qui s'avère être très pratique ;
- etc.

Vous pouvez obtenir la liste complète des *bundles* Symfony2 qui viennent de sortir sur <http://symfony2bundles.org/>, et il en existe déjà beaucoup. Imaginez donc dans quelques mois : tout ce que vous voudrez faire existera déjà en *bundle* ! Un blog ? Un *bundle*. Un forum ? Un *bundle* ! Et ainsi de suite.

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. 😊

La bonne pratique

Même si vous ne comptez pas forcément distribuer vos *bundles*, il est toujours bon de garder le découpage en *bundles* au sein de votre application et de ne pas tout mettre dans un seul *bundle* (c'est possible, oui). Pourquoi ? Parce que ça vous permet de découpler vos fonctionnalités et donc d'organiser naturellement votre code. Cela vous permet aussi de partager vos *bundles* entre deux projets.

La structure d'un bundle

Un *bundle* contient tout : routes, contrôleurs, vues, modèles, classes personnalisées, 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 :

Code : Autre - Sélectionner

```
Controller/      | Contient vos contrôleurs.
Entity/          | Contient vos modèles.
Form/            | Contient vos éventuels formulaires.
Resources/
-- config/       | Contient les fichiers de configuration de votre bundle (nous placerons les routes ici, par ex
-- public/       | Contient les fichiers publics de votre bundle : fichiers CSS et JavaScript, images, etc.
-- views/        | Contient les vues de notre bundle, les templates Twig.
Tests/           | Contient vos éventuels tests unitaires et fonctionnels. Nous développerons sans faire de tests
```

Retenez bien cette structure. Je vous guiderai pour chaque création de fichier, mais il est toujours bon de l'avoir en tête. 😊

Vous connaissez maintenant les grands principes de Symfony2, et c'est déjà beaucoup !

Dans le prochain chapitre, nous allons créer notre première page. Cela nous permettra de manipuler *bundles*, routes, contrôleurs et vues. Bref, nous mettrons en pratique tout ce que nous venons de voir en théorie.

Rendez-vous au prochain chapitre.

Mon premier Hello World avec Symfony2

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

Ce chapitre est un chapitre *express* : le but ici n'est pas de tout comprendre, mais de tout faire. Ainsi, à la fin du chapitre, vous aurez une page en état de fonctionnement, et ce sera dans les prochains chapitres que l'on détaillera précisément chaque module : **bundle**, **routeur**, **contrôleur** et **template**. On jouera avec pour mieux les maîtriser.

Ne bloquez donc pas sur un point si vous ne comprenez pas tout : c'est normal ! À la fin du chapitre, vous aurez une vision globale (mais floue) de la **création d'une page** et l'objectif sera

ne croquez donc pas sur un point si vous ne comprenez pas tout, c'est normal ! Lisez au chapitre, retournez aux liens proposés (dans tous) et revenez à une page et sujet bien atteint.

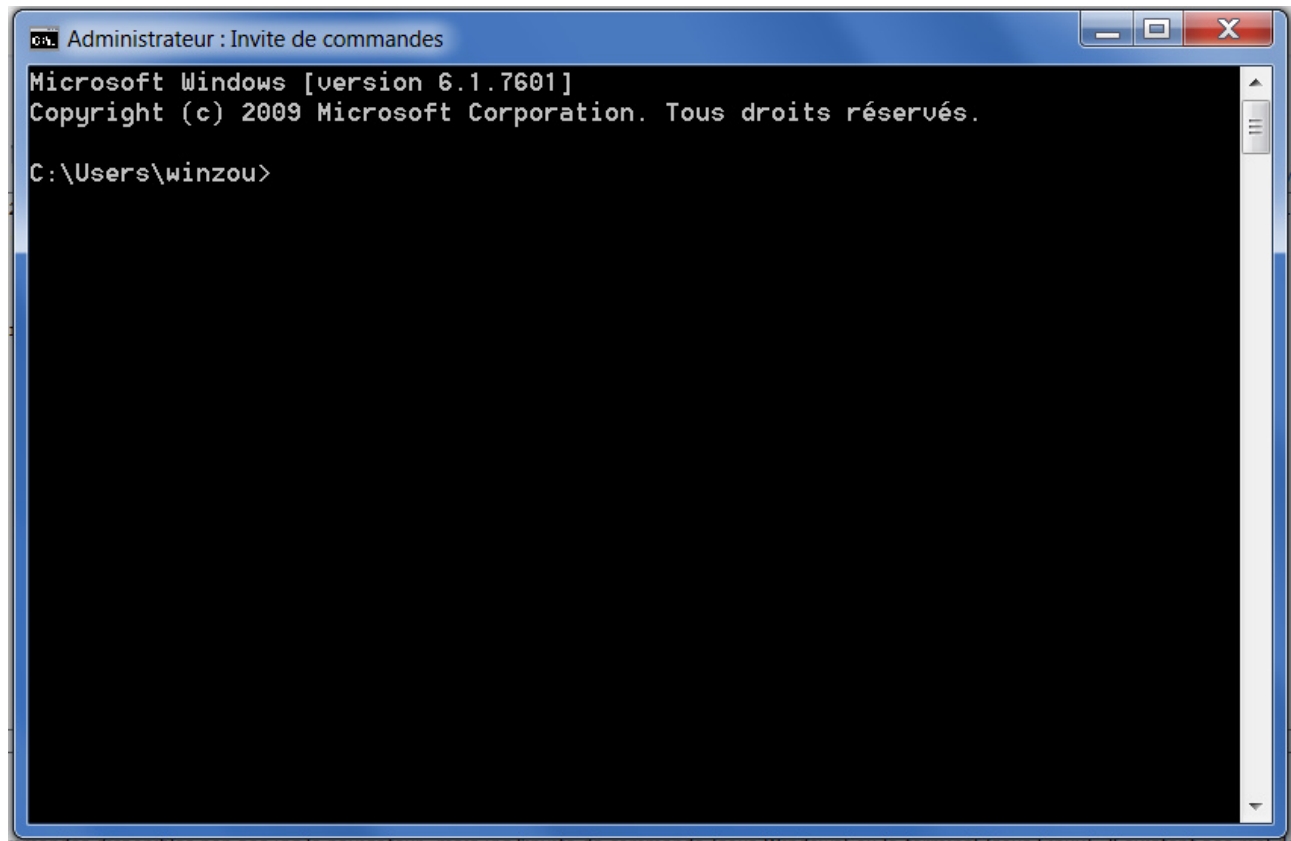
Bonne lecture !

Bien utiliser la console

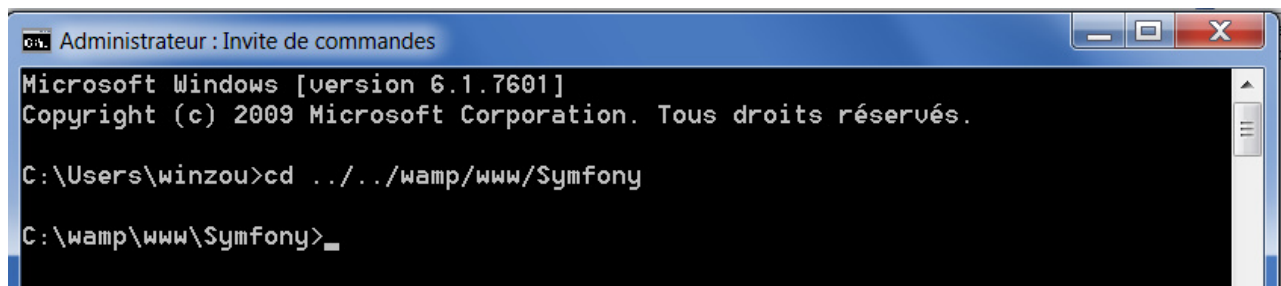
Tout d'abord, vous devez savoir une chose : Symfony2 intègre des commandes disponibles non pas *via* le navigateur, mais *via* l'invite de commande (sous Windows) ou le terminal (sous Linux). Il existe pas mal de commandes qui vont nous servir assez souvent lors du développement, apprenons donc dès maintenant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

Sous Windows

Lancez l'invite de commandes : Menu Démarrer > Programmes > Accessoires > Invite de commandes.



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



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

```
Administrator : Invite de commandes
D:\backup\www\asso>php app/console
Symfony version 2.0.0-RC6 - 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        -U Display this program version.
 --ansi           Force ANSI output.
 --no-ansi        Disable ANSI output.
 --no-interaction -n Do not ask any interactive question.
 --shell          -s Launch the shell.
 --env            -e The Environment name.
 --no-debug       Switches off debug mode.

Available commands:
 help                Displays help for a command
 list                Lists commands
 assetic
 assetic:dump        Dumps all assets to the filesystem
 assets
```

Et voilà, vous venez d'exécuter une commande Symfony !



La commande ne fonctionne pas ? Il vous dit que PHP n'est pas un exécutable ? Il faut que vous informiez Windows de l'emplacement exact de votre PHP en l'ajoutant dans la variable PATH, allez donc lire la FAQ PHP à ce sujet : <http://www.php.net/manual/fr/faq.installation.addtopath>

Sous Linux

Ouvrez le terminal. Placez vous dans le répertoire où vous avez mis Symfony2. Puis l'exécutable PHP est app/console, il faut donc lancer la commande `./app/console`. Je ne vous fais pas de capture d'écran, mais j'imagine que vous savez le faire !

Dans la suite du cours, j'utiliserai des commandes du type `php app/console la_commande` au lieu de `./app/console la_commande` car c'est la syntaxe pour nos amis sous Windows. Sachez juste l'adapter à chaque fois ! 😊



Amis linuxiens tant que je vous ai sous la main j'en profite. Symfony2 a déjà dû vous le dire lors du config.php, il a besoin d'avoir les droits suffisants pour écrire dans les répertoires app/cache et app/logs. Si vous ne l'avez pas déjà fait, autorisez votre httpd à écrire dans ces répertoires avec un `chmod -R 775 app/cache app/logs` par exemple.

A 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, une entité 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 ! Attaquons dès maintenant avec la commande `php app/console generate:bundle` !

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 !

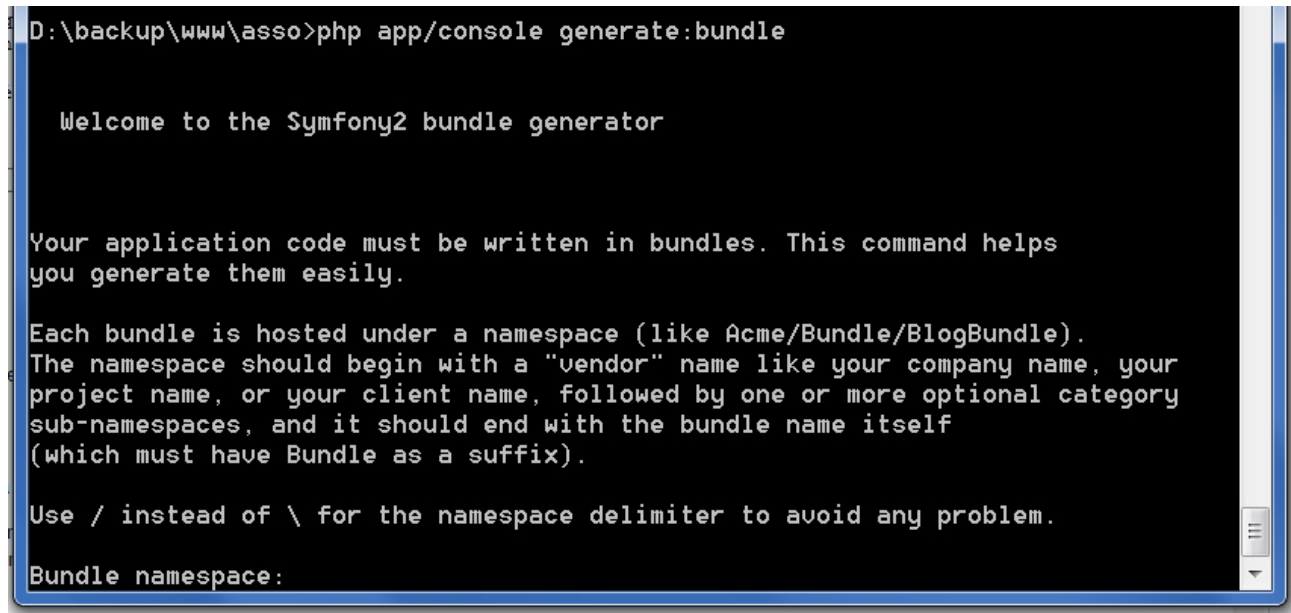
Exécuter la bonne commande

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 :



```
D:\backup\www\asso>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).

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. Nommons notre namespace « **SdzBlogBundle** », explications :

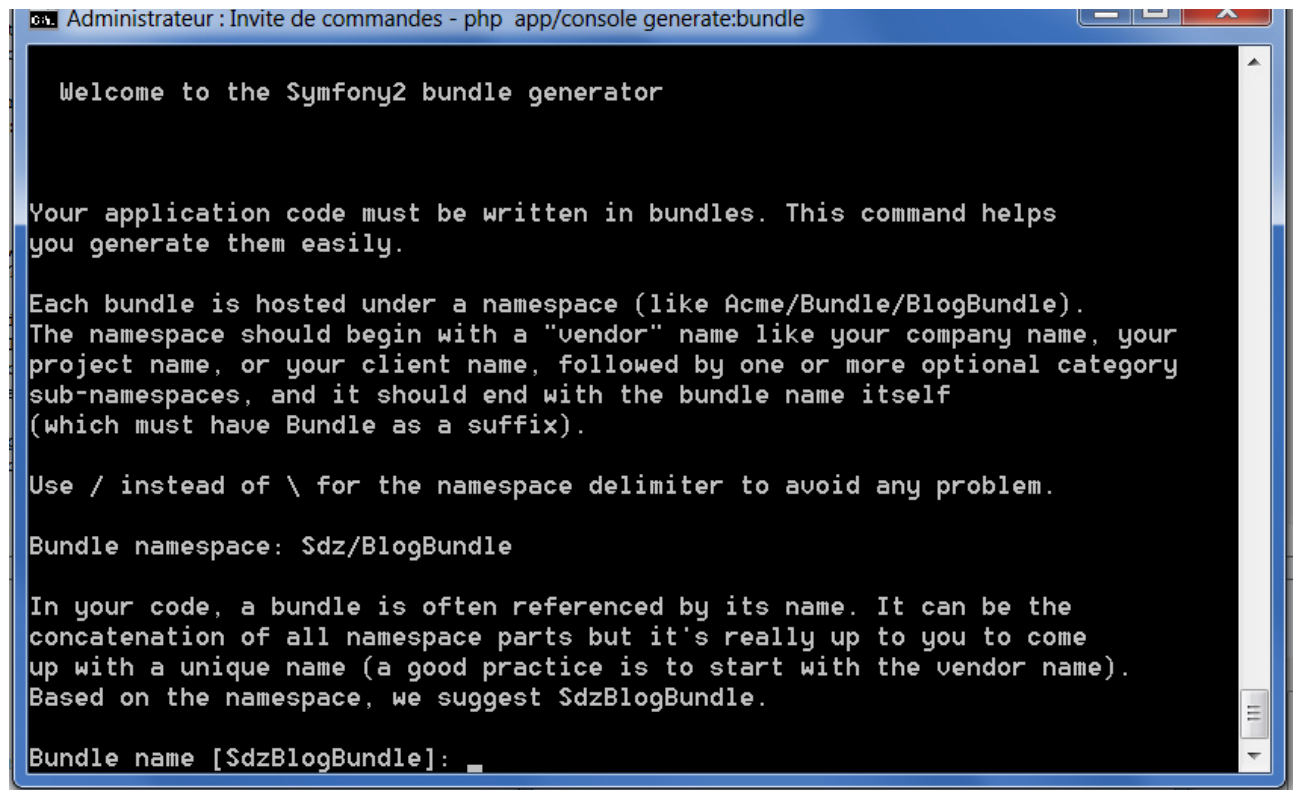
1. « Sdz » est le *namespace* racine : il vous représente. Vous pouvez mettre votre pseudo, le nom de votre site ou ce que vous voulez ;
2. « Blog » est le nom du *bundle* en lui-même : il définit ce que fait le *bundle*. Ici, nous créons un blog, nous l'avons donc simplement appelé « Blog » 🤖 ;
3. «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

Rentrez donc dans la console `Sdz/BlogBundle`.

2. Choisir le nom

Symfony2 vous demande le nom de votre bundle :





```
Administrateur : Invite de commandes - 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/Bundle).
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).

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

Bundle namespace: Sdz/BlogBundle

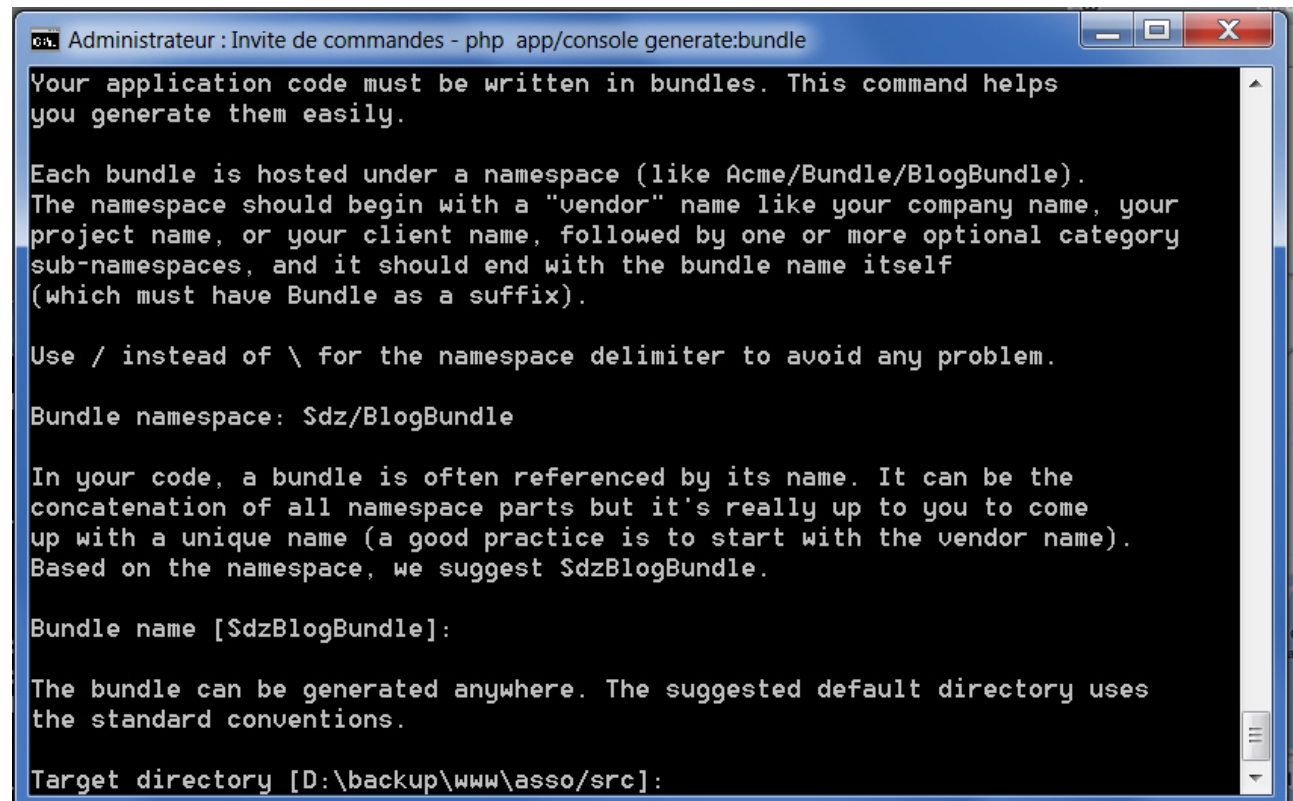
In your code, a bundle is often referenced by its name. It can be the
concatenation of all namespace parts but it's really up to you to come
up with a unique name (a good practice is to start with the vendor name).
Based on the namespace, we suggest SdzBlogBundle.

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, appuyez donc simplement sur "Entrée".

3. Choisir la destination

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



```
Administrateur : Invite de commandes - php app/console generate:bundle

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/Bundle).
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).

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

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]:

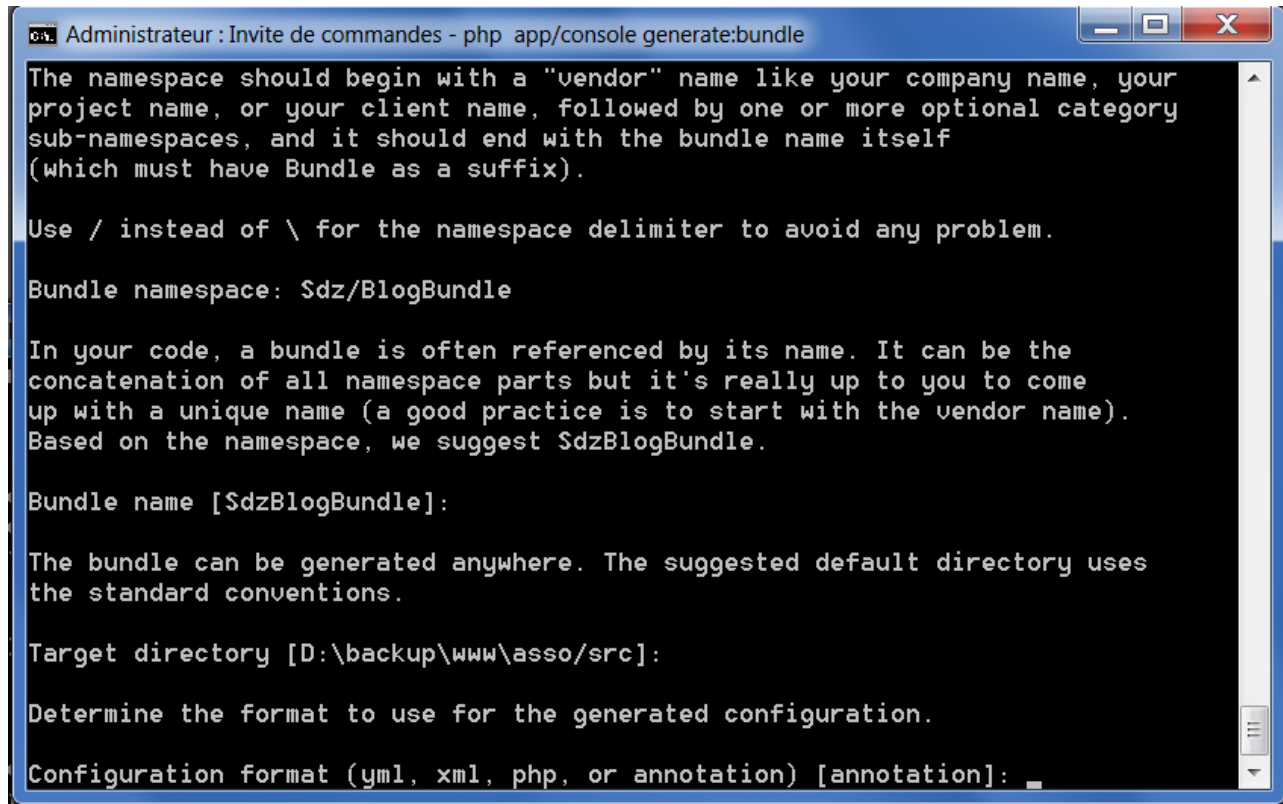
The bundle can be generated anywhere. The suggested default directory uses
the standard conventions.

Target directory [D:\backup\www\asso/src]:
```

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

4. Choisir le format de configuration

Symfony2 vous demande sous quelle forme vous voulez configurer votre bundle. Il existe plusieurs moyens comme vous pouvez le voir : Yaml, XML, PHP ou Annotations.



```
Administrateur : Invite de commandes - php app/console generate:bundle

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).

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

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]:

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

Target directory [D:\backup\www\asso/src]:

Determine the format to use for the generated configuration.

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

Chacune a ses avantages et inconvénients. Nous allons utiliser le Yaml ici, car est il bien adapté pour un bundle. Mais sachez que nous utiliseront les annotations pour nos futures entités par exemple. Entrez donc `yaml`.

5. Choisir quelle structure générer

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




```
GA. Administrateur : Invite de commandes - php app/console generate:bundle

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

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]:

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

Target directory [D:\backup\www\asso/src]:

Determine the format to use for the generated configuration.

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

To help you getting 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 :

```
GA. Administrateur : Invite de commandes

You are going to generate a "Sdz\BlogBundle\SdzBlogBundle" bundle
in "D:\backup\www\asso/src/" using the "yaml" 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!

D:\backup\www\asso>
```




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 !

Que s'est-il passé ?

Dans les coulisses, Symfony2 a fait pas mal de choses, revoyons tout ça à notre rythme. Allez dans le répertoire `src/Sdz/BlogBundle`. Voici donc ce que Symfony2 a généré tout seul.

La classe de base du bundle

Pour créer un bundle, le seul fichier obligatoire est en fait cette classe `SdzBlogBundle.php` à la racine du répertoire. Vous pouvez l'ouvrir et voir ce qu'il contient : pas très intéressant en soi; tant mieux que Symfony l'ait généré tout seul 🤖

En fait nous ne modifierons presque jamais ce fichier. Il ne sert que si vous voulez faire de l'héritage entre *bundles* (nous en reparlerons dans le chapitre traitant de la sécurité, vous avez le temps), ou si vous souhaitez utiliser des fonctionnalités avancées de Symfony2 dans votre *bundle*. Pour l'instant, nous ne le toucherons pas.

Controller/

Symfony2 a aussi créé un contrôleur de base, dans `Controller/DefaultController.php`. Ce contrôleur n'est qu'une base, un point de départ pour écrire notre propre code ensuite. Nous en reparlerons plus bas dans le paragraphe sur les contrôleurs.

DependencyInjection/

Dans ce répertoire, Symfony2 a créé les fichiers nécessaires pour configurer finement notre bundle. Comme on va commencer doucement, on ne se servira pas de ce répertoire pour le moment.

Resources/

Dans ce répertoire, deux fichiers sont plus importants pour l'instant.

Le fichier `Resources/config/routing.yml`, ouvrez-le, contient les routes du bundle. Nous le verrons plus en détails plus bas dans le paragraphe sur les routes.

Le fichier `Resources/views/Default/index.html.twig` correspond à la vue du contrôleur généré. Nous en reparlerons également plus bas, dans le paragraphe sur les vues.

Tests/

Dans ce répertoire, nous mettrons les tests de notre application. Nous les étudierons plus loin dans ce cours.

Conclusion

Nous aurons l'occasion d'utiliser ces fichiers créés tout au long du cours. Mais grâce au générateur de Symfony2, ils sont déjà présents et prêts à être utilisés !

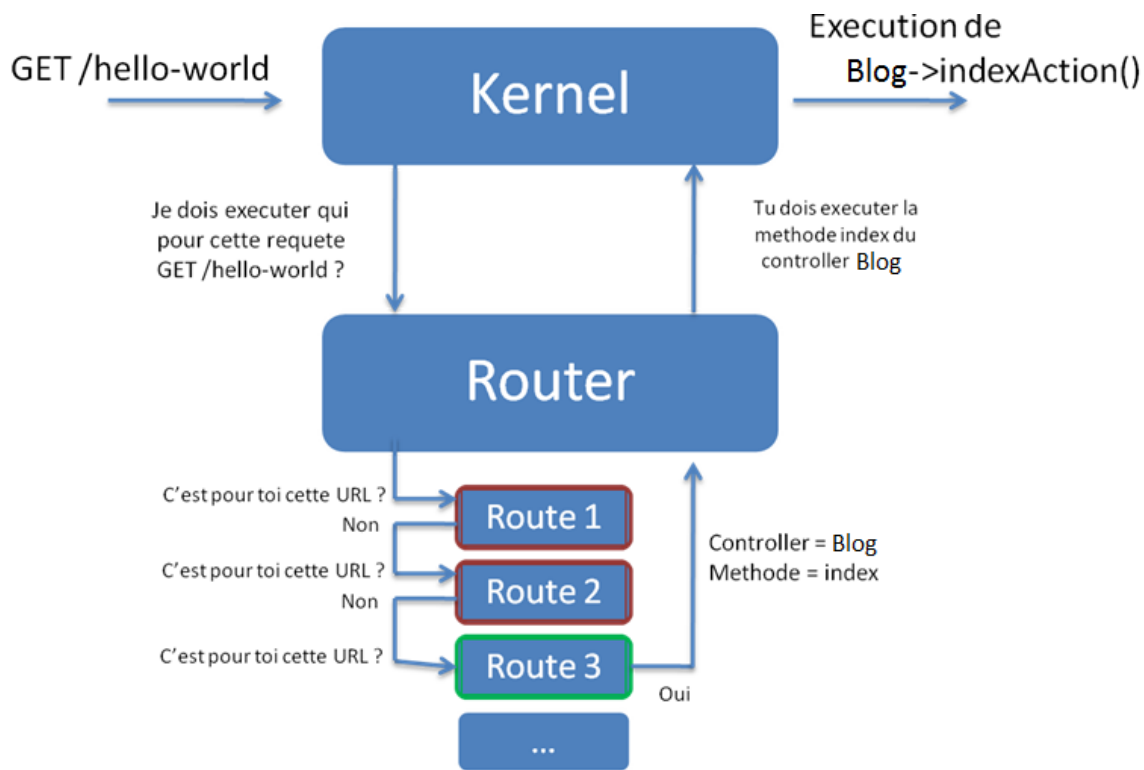
Créons notre route

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

Objectif

L'objectif ici est de dire à Symfony2 ce qu'il doit faire lorsque que l'on appelle l'URL `/hello-world`. 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.* ».

Voici un schéma de l'interaction Noyau - Routeur - Route :



Comme je l'ai dit, nous ne toucherons ni au noyau, ni au routeur : nous nous occuperons juste des routes. Ouf, voilà du travail en moins. 😊

1. Créons notre fichier de routes

Les routes se définissent dans un simple fichier texte, que Symfony2 a déjà généré. 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 : Autre - Sélectionner

```

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 en grand ici parce qu'un jour vous ferez l'erreur (l'inévitable ne peut être évité), et vous me remercirez de vous avoir mis sur la voie 😊 Et ceci est valable pour tous vos fichiers `.yml`

Essayons de comprendre rapidement ce fichier :

- le **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 » ;
- le **_controller** correspond à l'**action** que l'on veut exécuter et au **contrôleur** que l'on va appeler.

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 **pointillés** 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. Lorsque vous définissez cette méthode dans le contrôleur, vous devez y apposer le suffixe « Action », comme ceci : `<?php public function indexAction()` (nous le verrons plus loin dans ce chapitre).

2. Informons Symfony2 que nous avons des routes pour lui

Grâce au bon travail du générateur, Symfony2 est déjà au courant du fichier de routes de notre bundle. Mais sachez que ce n'est pas par magie ! Le comportement de Symfony 2 pour les routes est de n'ouvrir que le fichier `app/config/routing.yml`. Ouvrez donc ce fichier : outre les commentaires, vous voyez que le générateur a inséré une route spéciale qui va

importer le fichier de routes de notre bundle.

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.



C'est parce que ce fichier était vide (avant la génération du bundle) que l'on avait une erreur « page introuvable » en mode « prod » : comme il n'y a aucune route définie, Symfony2 nous informe à juste titre qu'aucune page n'existe.

Pour information, si le mode « dev » ne nous donnait pas d'erreur, c'est parce que le mode « dev » 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 des deux fichiers.

Revenons à nos moutons. En fait, on aurait pu ajouter notre route directement dans ce fichier `routing.yml`. Ça aurait fonctionné et ça 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.

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 `indexAction()` !

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 ». Les fichiers des contrôleurs doivent respecter une convention très simple : leur nom 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 pour nous, ce n'est qu'un exemple, on va utiliser le notre. Faites donc un simple copier-coller du `DefaultController.php` généré, et nommez-le `BlogController.php`. Ouvrez-le, et pour l'adapter à son nouveau nom, changez le nom de la classe de `DefaultController` à `BlogController`. On va aussi utiliser notre propre méthode `indexAction`, remplacez-la par la suivante :

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

// N'oubliez pas de rajouter ce use supplémentaire
use Symfony\Component\HttpFoundation\Response;

// Changez bien le nom du contrôleur
class BlogController extends Controller
{
    // Remplacez par cette méthode
    public function indexAction()
    {
        return new Response("Hello World !");
    }
}
```



Deuxième 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 3 : 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 5 : 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 11 : le nom de notre contrôleur respecte le nom du fichier ;
- ligne 14 : on définit la méthode `indexAction()`. N'oubliez pas de mettre le suffixe « **Action** » derrière le nom de la méthode ;
- ligne 16 : 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.

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 et un *template* en « langage Twig ».

PHP :

Code : PHP - [Sélectionner](#)

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

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

Et ce même *template*, mais avec Twig :

Code : HTML - [Sélectionner](#)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to Symfony!</title>
  </head>
  <body>
    <h1>{{ page_title }}</h1>

    <ul id="navigation">
      {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</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) ?>`, vous êtes d'accord ?

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 généré par Symfony2. Appelons notre

`template Blog/index.html.twig` et mettons-le dans le repertoire des vues. Nous avons donc le fichier `src/Sdz/BlogBundle/Resources/views/Blog/index.html.twig`.



Blog/index.html.twig ? Découpons ce nom :

- 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 du code PHP.

Revenez à notre *template* et copiez-collez ce code à l'intérieur :

Code : HTML - [Sélectionner](#)

```
<!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 `<?php indexAction() ?>` que nous allons appeler le *template*. Cela se fait très simplement avec la méthode `<?php $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 - [Sélectionner](#)

```
<?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 12. La convention pour le nom du *template* est la même que pour le nom du contrôleur, souvenez-vous.

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



Notez également l'apparition de la *toolbar* en bas de la page. Symfony2 l'ajoute automatiquement lorsqu'il détecte la balise fermante `</html>`. C'est pour cela que nous ne l'avons pas tout à l'heure avec notre « Hello World » tout simple. 😊

Voulez-vous vous amuser un peu avec les variables Twig ?

- Modifiez la ligne 12 du contrôleur pour rajouter un 2^e argument à la méthode `render()` : `<?php return $this->render('SdzBlogBundle:Blog:index.html.twig', array('nom' => 'winzou'));`.
- Modifiez votre *template* en remplaçant `<h1>Hello World !</h1>` par `<h1>Hello {{ nom }} !</h1>`.
- C'est tout ! Rechargez la page. Bonjour à vous également. 😊

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 devons saisir notre nom lorsque nous rédigerons 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 `Default` ;
- Les vues dans le répertoire `Default` ;
- La route `SdzBlogBundle_homepage`.

Rafraîchissez la page pour vérifier que tout est bon... raté ! 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). Deux cas de figure :

- En mode prod, c'est simple, Symfony2 ne vide jamais le cache. Ca lui permet de ne faire aucune vérification sur la validité du cache, 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 apparaître des bugs parfois, comme l'erreur filemtime que vous avez là. 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, des fois 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.

Evidemment, 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, c'est pas un `cache:clear` qui va la résoudre ! 🤔

Et maintenant, c'est parti !

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. La création du *bundle*, par exemple, est faite une fois pour toutes. Le contrôleur peut servir plusieurs pages différentes. Le fichier *routing* est déjà notifié à Symfony2. Le *template* peut être copié-collé, etc. Si la création de notre première page était un peu longue, la création d'une deuxième sera très rapide. D'ailleurs, 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 !

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-systeme-de-route` est bien plus sexy que `index.php?controlleur=article&methode=voir&id=5` !



Vous avez peut-être 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 `yourBundle/Resources/config/routing.yml`, contient la définition des routes. Chaque route fait la correspondance entre une **URL** et le contrôleur à appeler. Pour vous donner un exemple, voici ce que l'on peut voir dans ce fichier :

Code : Autre - [Sélectionner](#)

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

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

liste_articles:
  pattern:  /blog/liste/{page}
  defaults: { _controller: SdzBlogBundle:Blog:liste, page: 1 }
  requirements:
    page:  \d+
```



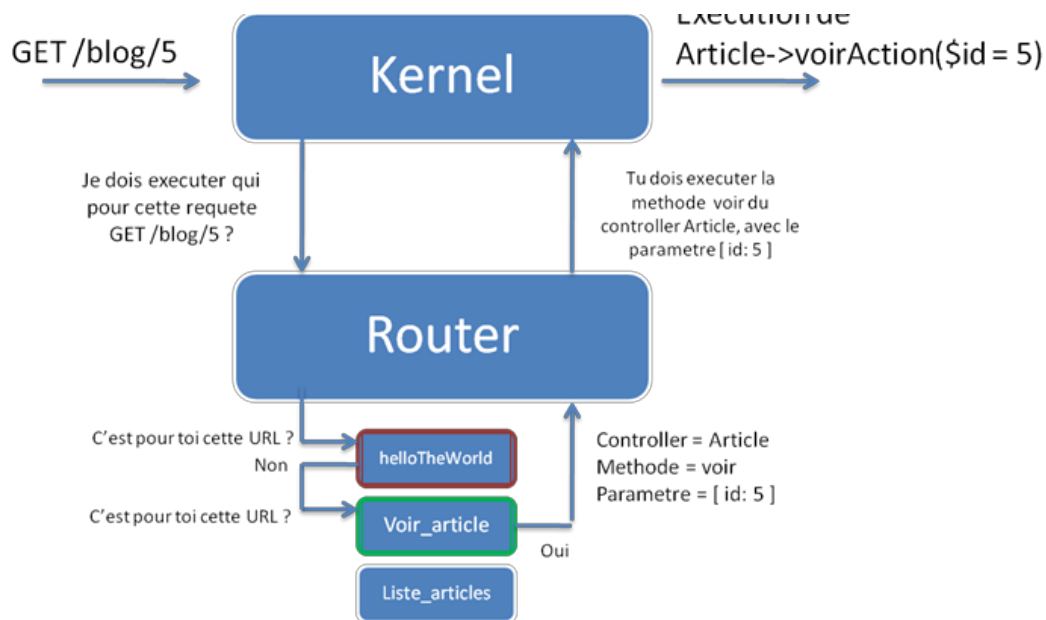
Petit rappel au cas où : l'indentation se fait avec 4 espaces par niveau, et non avec des tabulations 😊

Fonctionnement du routeur

Dans l'exemple ci-dessus, vous pouvez distinguer trois blocs. Ce sont les routes. Nous les verrons en détail plus loin, mais vous pouvez constater que chaque route prend :

- une entrée (ligne *pattern*) : c'est l'**URL** à transformer ;
- une sortie (ligne *defaults*) : c'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. Voici le schéma équivalent au chapitre précédent, mais actualisé pour notre fichier de route juste au-dessus :



Et voici en texte le fonctionnement, pas à pas :

- on appelle l'*URL* `/blog/5` ;
- le routeur essaie de faire correspondre cette *URL* avec la *pattern* de la première route. Ici, `/blog/5` ne correspond pas du tout à `/blog` (ligne *pattern* de la première route) ;
- le routeur passe donc à la route suivante. Il essaie de faire correspondre `/blog/5` avec `/blog/{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 :
 - `/blog (URL) = /blog (route)`,
 - `5 (URL) = {id} (route)` ;
- le routeur s'arrête donc, il a trouvé sa route ;
- 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`.* » ;
- le routeur renvoie donc ces informations au *kernel* (le noyau de Symfony2) ;
- le noyau va exécuter le bon contrôleur !

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

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 : « Va voir dans le répertoire de ce *bundle* », répertoire que vous lui avez indiqué dans le fichier `app/autoload.php`. 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 la méthode à exécuter au sein du contrôleur. 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 : Autre - Sélectionner

```
helloTheWorld:
    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 :

- « **helloTheWorld** » est le nom de la route. Il n'a aucune importance lors de la transformation *URL* ? contrôleur, 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 faire faire le travail à notre place ! Retenez donc pour l'instant qu'il faut qu'un nom soit unique et clair ;
- « **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 !

vous avez maintenant les bases pour créer une route simple :

Créer une route avec des paramètres

Reprenons la deuxième route de notre exemple :

Code : Autre - [Sélectionner](#)

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

Grâce au paramètre `{id}` dans le *pattern* de notre route, toutes les URL du type `/blog/*` seront gérées par cette route, par exemple : `/blog/5` ou `/blog/654`, ou même `/blog/sodfihsodfi`. Par contre, l'URL `/blog` ne sera pas interceptée car le paramètre `{id}` n'a pas été 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. La puissance réside dans le fait que ce paramètre `{id}` est accessible depuis votre contrôleur ! Si vous appelez l'URL `/blog/5`, alors depuis votre contrôleur, vous aurez la variable `<?php $id ?>` (du nom du paramètre) qui aura pour valeur « 5 ». Voici un exemple concret du contrôleur correspondant :

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    // On définit la méthode avec un argument $id pour aller avec la route correspondante.
    public function voirAction($id) // Regardez notre route, c'est la méthode « voir » qui doit
    être appelée.
    {
        // $id vaut 5 si l'on a appelé l'URL /blog/5.

        // On récupère depuis la base de données l'article correspondant à l'$id.
        // C'est normal si vous ne comprenez pas cette ligne, admettez juste qu'elle récupère
        l'article. ;)
        $article = $this->get('blog.articles.manager')->getArticle(array('id' => $id));

        // On fait appel au template en lui passant notre article.
        return $this->render('SdzBlogBundle:Blog:voir.html.twig', array(
            'article' => $article,
        ));
    }
}
```

Vous pouvez bien sûr multiplier les paramètres au sein d'une même route. Rajoutez cette route :

Code : Autre - [Sélectionner](#)

```
voir_article_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 `/article/2011/symfony.xml`, etc.

Adaptons notre contrôleur pour récupérer tous les paramètres :

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    // On a rajouté les paramètres $annee et $format.
    public function voirSlugAction($slug, $annee, $format)
    {
        // Etc. Le contenu de la méthode ressemble à celui de voirAction().
    }
}
```



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.etait.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. ».

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 :

Code : [Autre](#) - [Sélectionner](#)

```
voir_article_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 : [Autre](#) - [Sélectionner](#)

```
voir_article_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.

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

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

Code : [Autre](#) - [Sélectionner](#)

```
voir_article_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 `<?php $format ?>` comme avant et celui-ci vaudra « html ».

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. Vous pouvez récupérer ce format via la méthode `getRequestFormat()` de l'objet **Request**. Par exemple, depuis un contrôleur : `<?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 - [Sélectionner](#)

```
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 plus haut 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, imaginez que nous avons une route nommée « voir_article » qui écoute sur l'*URL* `/blog/{id}`. Vous décidez un jour de rallonger vos *URL* et vous aimeriez bien que vos articles soient disponibles depuis `/blog/voir/{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 - [Sélectionner](#)

```
<?php
namespace Sdz\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
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('voir_article', array('id' => $id));
        // $url vaut "/blog/5" (ou "/blog/voir/5" si vous avez modifié votre route).

        // 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 - [Sélectionner](#)

```
<?php
```

```
$url = $this->generateUrl('voir_article', array('id' => $id), true);
```

Ainsi, `<?php $url ?>` vaut « <http://monsite.com/blog/5> » et pas uniquement « /blog/5 ».

2. Depuis une vue Twig

Mais 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 - [Sélectionner](#)

```
<a href="{{ path('voir_article', { '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 : Autre - [Sélectionner](#)

```
sdzblog:
  pattern:    /{page}
  defaults:   { _controller: SdzBlogBundle:Blog:liste, page: 1 }
  requirements:
    page:     \d+
```

Page de visualisation d'un article

Afin de favoriser le référencement, il est très intéressant de mettre dans l'*URL* non pas l'id de l'article, mais son titre. On utilise dans ce genre de cas le « slug » du titre, c'est-à-dire une version épurée du titre compatible avec les *URL* : on enlève les caractères spéciaux, les espaces, etc. Par exemple, « Mon weekend de folie !! » deviendra « mon-weekend-de-folie ».

Voici la route :

Code : Autre - [Sélectionner](#)

```
sdzblog_voir:
  pattern:    /{slug}
  defaults:   { _controller: SdzBlogBundle:Blog:voir }
```



Attention : du fait de l'ordre des routes, vous ne pourrez pas avoir un article ayant pour titre un nombre seulement. Par exemple, si vous appelez votre article « 6 », pour accéder à cet article, vous irez sur `/blog/6...` mais la première route va intercepter cette *URL* et vous tomberez sur la page 6 de la liste de vos articles !

Ajout, modification et suppression

Les routes sont simples :

Code : Autre - [Sélectionner](#)

```
sdzblog_ajouter:
  pattern:    /ajouter/article
  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+

```



Nous avons esquivé les problèmes de titre en rajoutant à chaque fois une partie `/article` ou `{id}`. Ainsi, vous pourrez avoir un article ayant pour titre « Ajouter », cela ne posera pas de problème. 😊

Récapitulatif

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

Code : [Autre](#) - [Sélectionner](#)

```

sdzblog:
  pattern: /{page}
  defaults: { _controller: SdzBlogBundle:Blog:liste, page: 1 }
  requirements:
    page: \d+

sdzblog_voir:
  pattern: /{slug}
  defaults: { _controller: SdzBlogBundle:Blog:voir }

sdzblog_ajouter:
  pattern: /ajouter/article
  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 : [Autre](#) - [Sélectionner](#)

```

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

```

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 !

Les contrôleurs avec Symfony2

Ah, le **contrôleur** ! Notre ami !

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, 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 !

Bonne lecture !

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 langage HTTP. Il existe dans Symfony2 une classe **Response**. Retourner une réponse signifie donc tout simplement : instancier un objet **Response**, disons `<?php $response ?>` et faire un `<?php return $response ?>`.

Voici le contrôleur le plus simple qui soit, c'est le contrôleur que l'on a créé dans les chapitres précédents. Il dispose d'une seule méthode, nommée « index », et retourne une réponse qui ne contient que : « Hello World ! ».

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Controller;

// On utilise le namespace du contrôleur de base de Symfony, car notre contrôleur
// va hériter de celui-là.
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

// On utilise le namespace de la classe Response.
use Symfony\Component\HttpFoundation\Response;

class BlogController extends Controller
{
    public function indexAction()
    {
        $response = new Response('Hello World !');

        return $response;
    }
}
```

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 ou bien le nom d'un membre à chercher dans la base de données, etc. Ces paramètres, nous savons déjà les gérer, nous l'avons vu dans le chapitre sur le routeur. Mais voici un petit rappel.

La méthode qui esquivé l'objet « Request »

Tout d'abord côté route. Nous avons déjà la route **sdzblog_voir** :

Code : Autre - [Sélectionner](#)

```
sdzblog_voir:
    pattern:  /{slug}
    defaults: { _controller: SdzBblogBundle:Blog:voir }
```

Ensuite, côté contrôleur. Pour récupérer ce paramètre `{slug}`, créons l'action « voir » dans notre contrôleur ayant pour paramètre `<?php $slug ?>` :

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Controller;

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

```

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

```

Créons également le *template*, car celui-ci n'existe pas encore. Dans le fichier `src/Sdz/BlogBundle/Resources/views/Blog/voir.html.twig`, copiez-collez ce code :
Code : HTML - [Sélectionner](#)

```

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

```

Vous pouvez maintenant vous amuser à consulter l'adresse http://localhost/Symfony/web/app_dev.php/_li mier-weekend mais encore http://localhost/Symfony/web/app_dev.php/_li y2-c-est-cool, etc.

La méthode qui utilise l'objet « Request »

Vous avez vu : dans la première méthode, nous n'avons pas utilisé du tout l'objet **Request**. Du moins... pas explicitement ! En fait, Symfony2 s'en est bien servi, mais sans nous le dire. Si par exemple, nous avons une *URL* du type `/blog/mon-dernier-weekend?tag=vacances`, il nous faut bien un moyen pour récupérer ce paramètre « tag » !

Tout d'abord, récupérons l'objet **Request** depuis notre contrôleur : `<?php $request = $this->get('request');`. Voilà, c'est aussi simple que ça. La méthode `<?php get()`, nous en reparlerons, permet d'accéder à toute sorte de service, dont notre requête. Maintenant que nous l'avons, récupérons nos paramètres :

Code : PHP - [Sélectionner](#)

```

<?php
namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function voirAction($slug)
    {
        // On récupère la requête.
        $request = $this->get('request');

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

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

```

Nous avons utilisé `$request->query` pour récupérer les paramètres de l'*URL* passés en « GET ». Mais nous aurions pu utiliser `$request->cookies` pour les cookies, ou bien `$request->request` pour les paramètres « POST ».

Pour faire apparaître la variable `tag` dans notre *template*, modifions-le :

Code : HTML - [Sélectionner](#)

```

<!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>
            Ici nous pourrions lire l'article ayant comme slug : {{ slug }}<br />
            Cet article pourrait avoir comme tag « {{ tag }} »<br />

```

```
        Mais pour l'instant, nous ne savons pas encore le faire, cela viendra !
    </p>
</body>
</html>
```

Et vous n'avez plus qu'à tester le résultat bien sûr : http://localhost/Symfony/web/app_dev.php/?tag=vacances

Les autres méthodes de l'objet « Request »

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()` pour cela :

Code : PHP - [Sélectionner](#)

```
<?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 - [Sélectionner](#)

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

Récupérer la langue préférée du visiteur

Si votre site est multilingue, ou si vous avez au moins une ressource (fichier, page, etc.) à proposer en plusieurs langues, vous avez besoin de connaître la langue préférée de votre visiteur. Il faut dans ce cas utiliser la méthode `<?php $request->getPreferredLanguage()` :

Code : PHP - [Sélectionner](#)

```
<?php
// Récupérez la langue préférée.
$langue1 = $request->getPreferredLanguage();

// Ou alors, plus fréquent, vous récupérez la langue préférée de votre visiteur
// parmi les langues disponibles que vous avez. Exemple :
$langue2 = $request->getPreferredLanguage(array('fr', 'en'));
```

Dans ce dernier exemple, vous avez, disons, un fichier disponible uniquement en français et en anglais. Mais si un visiteur russe arrive sur votre site, sa langue préférée dans l'absolu est... le russe ! `<?php $langue1 ?>` vous retournera donc le russe, mais ça ne vous avance pas beaucoup. Dans ce cas, `<?php $langue2 ?>` vous retournera sûrement l'anglais, car j'imagine que la 2^e langue des Russes est plus l'anglais que le français.

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 : <http://api.symfony.com/2.0/Symfony/Component/Request.html>. 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 derrière, voici la manière longue et décomposée de construire et de retourner une réponse :

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

// ...
```



```

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

class BlogController extends Controller
{
    public function voirAction($slug)
    {
        // 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\[...\]chain-weekend](http://localhost/Symfony/web/app_dev.php[...]chain-weekend) si vous avez gardé les mêmes routes depuis le début.

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

Réponses et vues

Généralement, vous préférerez que votre réponse soit contenue dans une vue, dans une *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 - [Sélectionner](#)

```

<?php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function voirAction($slug)
    {
        // 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(
            'slug' => $slug,
            'tag' => 'Politique'
        ));
    }
}

```

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 derrière les coulisses, nous n'avons pas à le manipuler directement.

Si vous ne deviez retenir qu'une seule chose de ce paragraphe, 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 :

Code : PHP - [Sélectionner](#)

```

<?php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function voirAction($slug)
    {
        // On utilise la méthode « generateUrl() » pour obtenir l'URL d'un article, par exemple.
        return $this->redirect($this->generateUrl('sdzblog_voir', array('slug' => 'weekend-au-
        cambodge')));
    }
}

```

```
}
```

Changer le « Content-type » de la réponse

Lorsque vous retournez autre chose que du HTML, il faut que vous changiez 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 - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Controller;

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

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

        // 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\[...\]/mes-vacances](http://localhost/Symfony/web/app_dev.php[...]/mes-vacances).

Les 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.

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 - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Controller;

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

class BlogController extends Controller
{
    public function voirAction($slug)
    {
        // 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);
    }
}
```

```

        $mailer->send($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');
    }
}

```



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

Brève liste des services

Évidemment, vous savez récupérer un service, mais encore faut-il connaître leur nom ! Et savoir les utiliser ! Ci-dessous 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 - [Sélectionner](#)

```

<?php
// ...
public function voirAction($slug)
{
    // 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->has('MonTemplate') ?>` qui permet de vérifier si « MonTemplate » existe ou non.

Request

Eh oui, encore elle. C'est également un service ! Vous pouvez donc la récupérer via `<?php $this->get('request') ?>`.

Session

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

Code : PHP - [Sélectionner](#)

```

<?php
// ...
public function voirAction($slug)
{
    // 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');
}

```

Doctrine

Si vous ne savez pas de quoi il s'agit, c'est normal. C'est le service qui gère vos modèles ! Évidemment, un chapitre entier lui est consacré. Mais voici juste de quoi avoir une idée sur le moyen de récupérer l'utilisateur d'id 5, par exemple :

Code : PHP - [Sélectionner](#)

```

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

    // On récupère le sous-service qui permet de gérer les utilisateurs.
    $gestion_utilisateurs = $doctrine->getEntityManager()-
>getRepository('Sdz\BonjourBundle\Entity\Utilisateur');

    // Récupération de l'utilisateur d'id 5.
    $utilisateur_5 = $gestion_utilisateurs->find(array('id' => 5));

    // On n'oublie pas de renvoyer une réponse.
    return new Response('Le pseudo de l'utilisateur 5 est : '.$utilisateur_5->getPseudo());
}

```

N'essayez pas d'exécuter ce code, il ne fonctionnera pas ! Nous n'avons pas encore défini nos entités **Doctrine**. Nous le ferons dans un prochain chapitre.

Vous pouvez malgré tout voir dans cet exemple que le contrôleur ne sait absolument pas comment sont gérés les utilisateurs : il ne fait que se servir du service **Doctrine** et de ses sous-services. Cela permet d'avoir un code très clair dans le contrôleur : aucune requête SQL, etc.

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 2, 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 pourrions 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 : Autre - [Sélectionner](#)

```

sdzblog:
  pattern:    /{page}
  defaults:  { _controller: SdzBlogBundle:Blog:liste, page: 1 }
  requirements:
    page: \d+

sdzblog_voir:
  pattern:    /{slug}
  defaults:  { _controller: SdzBlogBundle:Blog:voir }

sdzblog_ajouter:
  pattern:    /ajouter/article
  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 (src/Sdz/BlogBundle/Controller/BlogController.php):

Code : PHP - [Sélectionner](#)

```

<?php

namespace Sdz\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

```

```

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

// //! Attention, nouvel objet à charger !
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class BlogController extends Controller
{
    public function listeAction($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( $page < 1 )
        {
            // On déclenche une exception NotFoundHttpException, cela va afficher
            // la page d'erreur 404 (on pourra personnaliser cette page plus tard, d'ailleurs).
            throw new NotFoundHttpException('Page inexistante (page = '.$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.
        // Ce template n'existe pas encore, on va le créer dans le prochain chapitre.
        return $this->render('SdzBlogBundle:Blog:liste.html.twig');
    }

    public function voirAction($slug)
    {
        // Ici, on récupérera l'article correspondant au $slug.
        return $this->render('SdzBlogBundle:Blog:voir.html.twig');
    }

    public function ajouterAction()
    {
        // Ici, on s'occupera de la création et de la gestion du formulaire (via un service).

        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 (via un service).

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

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

        // Ici, on gèrera 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 ou 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](#).

N'oubliez pas d'ajouter le `<?php use ?>` correspondant au début du fichier pour pouvoir utiliser l'objet `NotFoundHttpException`.

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, aucune des actions ne va 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 d'accueil, à l'adresse http://localhost/Symfony/web/app_dev.php/blog. 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 :

Citation

```
[2011-06-06 16:01:16] request.INFO: Matched route "sdzblog" (parameters: " controller": "SdzBlogBundle\Controller\BlogController::listeAction", "page": "1", " route":
```

```
"sdzblog")
```

On voit que c'est bien la bonne route qui est utilisée, super ! On voit aussi que le paramètre « page », non défini, est bien mis à 1 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*, nous avons bien : « *BlogController::listeAction[sdzblog]404* ». Il se lit comme cela :

- `BlogController::listeAction` : c'est le contrôleur et sa méthode utilisée ;
- `sdzblog` : c'est le nom de la route utilisée ;
- `404 =>` c'est le code HTTP retourné. Ici, 404 correspond bien à notre « page introuvable ».

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, dès les premiers chapitres de la partie 2, vous apprendrez tout ce qu'il faudra pour commencer à coder à l'intérieur de vos contrôleurs. 😊
En attendant, rendez-vous au prochain chapitre pour en apprendre plus sur les *templates*.

Le moteur de template Twig

Les *templates* vont nous permettre de séparer le code PHP du code HTML/XML/Text/, etc. Intéressé ? Lisez la suite. 😊

Les templates Twig

Intérêt

Les *templates* 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 design fait du HTML, il n'a pas à subir votre code barbare PHP auquel il ne comprend rien.

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 `<?php htmlentities() ?>` ou `<?php 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, le contenu de l'e-mail sera placé 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 appeler un certain *template* en lui transmettant certaines variables :

Code : PHP - [Sélectionner](#)

```
<?php
// À la fin d'une action, pour afficher le template :
return $this->render('SdzBlogBundle:Blog:index.html.twig', array(
    'var1' => $var1,
    'var2' => $var2
));

// Au milieu d'une action, pour récupérer le contenu d'un template :
$contentu = $this->renderView('SdzBlogBundle:Blog:email.txt.twig', array(
    'var1' => $var1,
    'var2' => $var2
));
// Puis on envoie l'e-mail
mail('moi@siteduzero.com', 'Inscription OK', $contentu);
```

Et le `template SdzBlogBundle:Blog:email.txt.twig` se trouve dans `src/Sdz/BlogBundle/Resources/views/Blog/email.txt.twig`. Il contient par exemple :

Code : Autre - [Sélectionner](#)

```
Bonjour {{ pseudo }},  
Toute l'équipe du site se joint à moi pour vous souhaiter la bienvenue sur notre site !  
Revenez nous voir souvent !
```

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

- d'abord, vous donner les outils pour faire un code dynamique de base : savoir faire des boucles, des conditions, appliquer des filtres aux variables, etc. ;
- ensuite, 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.).

La syntaxe de base

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.

Variable

Afficher une variable en appliquant un filtre ou non se fait avec les doubles accolades « `{{ ... }}` ». Voici quelques exemples.

Afficher une variable simple :

Code : Autre - [Sélectionner](#)

```
Pseudo : {{ pseudo }}
```

Afficher l'index d'une variable de type **Array** (un tableau) :

Code : Autre - [Sélectionner](#)

```
Identifiant : {{ user['id'] }}
```

Utiliser un filtre. Ici, « `upper` » met tout en majuscules :

Code : Autre - [Sélectionner](#)

```
Pseudo en lettres majuscules : {{ pseudo|upper }}
```

Combiner les filtres. Ici, « `striptags` » supprime les balises et « `title` » met la première lettre de chaque mot en majuscule. Remarque : les filtres s'appliquent de gauche à droite. Ici `striptags` va s'appliquer au texte, puis `title` sera appliqué. L'équivalent PHP serait (en supposant que les fonctions existent) : `title(striptags($news['texte']))`.

Code : Autre - [Sélectionner](#)

```
Message : {{ news['texte']|striptags|title }}
```

Utiliser un filtre avec des arguments (attention ici, il faut que `news['date']` soit un objet de type **Datetime**) :

Code : Autre - [Sélectionner](#)

```
Date : {{ news['date']|date('d/m/Y') }}
```

Expressions

De l'autre côté, vous avez les structures dont la syntaxe ressemble à « `{% ... %}` ». Voici quelques exemples.

Condition simple :

Code : Autre - [Sélectionner](#)

```
{% if pseudo == 'winzou' %}  
    OK accès autorisé  
{% endif %}
```

Boucle simple :

Code : [HTML - Sélectionner](#)

```
<ul>
  {% for user in users %}
    <li>{{ user['pseudo'] }}</li>
  {% endfor %}
</ul>
```

Définition d'une variable depuis le *template* :

Code : [Autre - Sélectionner](#)

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

Les filtres utiles

Upper

{{ var|upper }} met toutes les lettres de {{ var }} en majuscules.

Striptags

{{ var|striptags }} supprime tous les tags XML de {{ var }} (et supprime les espaces consécutives également).

Date

{{ var|date('d/m/Y') }} formate la date {{ var }} suivant le format donné en argument. La variable {{ var }} peut valoir tout ce que **Datetime** peut prendre en argument. Par exemple, ce code affichera la date d'aujourd'hui : {{ "now"|date('d/m/y') }}.

Format

{{ "Il y a %s pommes et %s poires dans le jardin"|format(153, nb_poires) }} remplace %s par les arguments tout comme [printf](#) le fait.

Length

{{ var|length }} retourne le nombre d'éléments du tableau si {{ var }} est un tableau, et le nombre de caractères si {{ var }} est une chaîne de caractères.

Les autres

Il en existe d'autres, et vous pourrez aussi en créer vous-même. La liste exhaustive se trouve dans la documentation de Twig accessible à cette adresse : <http://www.twig-project.org/doc/templates#lt-in-filters>.

Les expressions utiles

For

Parcourir un tableau :

Code : [HTML - Sélectionner](#)

```
<ul>
  {% for user in users %}
    <li>{{ user['pseudo'] }}</li>
  {% endfor %}
</ul>
```

Parcourir un tableau et avoir accès aux clés :

Code : [HTML - Sélectionner](#)

```
<ul>
  {% for key, user in users %}
    <li>{{ user['pseudo'] }} (id: {{ key }})</li>
  {% endfor %}
</ul>
```

Parcourir un tableau et gérer le cas où le tableau serait vide (raccourci bien pratique) :

Code : [HTML - Sélectionner](#)


```
<ul>
  {% for user in users %}
    <li>{{ user['pseudo'] }}</li>
  {% else %}
    <li><em>Aucun utilisateur</em></li>
  {% endfor %}
</ul>
```

If

(Exemple repris de la documentation, il me fait marrer.)

Code : Autre - [Sélectionner](#)

```
{% if kenny['sick'] %}
  Kenny is sick.
{% elseif kenny['dead'] %}
  You killed Kenny! You bastard!!!
{% else %}
  Kenny looks okay --- so far
{% endif %}
```

Les tests utiles

Is defined

Pour vérifier qu'une variable existe :

Code : Autre - [Sélectionner](#)

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

Is even / Is odd

Pour vérifier qu'un nombre est pair (even) ou impair (odd) :

Code : Autre - [Sélectionner](#)

```
{% if foo is even %}
  ...
{% endif %}
```

Les autres

La liste complète se trouve aussi dans la documentation : [http://www.twig-project.org/doc/templa \[...\] uilt-in-tests](http://www.twig-project.org/doc/templa [...] uilt-in-tests).

La sécurité avant tout !

Dans tous les exemples précédents, vos variables ont été protégées par Twig ! Twig applique par défaut un filtre sur toutes les variables que vous affichez, afin de 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 affiché est en réalité "mon<pseudo" au lieu de "mon<pseudo". Très pratique ! Et donc à savoir : inutile de protéger vos variables en amont, Twig s'occupe de tout en fin de chaîne !

Dans le cas où vous voulez afficher volontairement une variable qui contient du HTML, 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, quelque soit ce qu'elle contient.

Hériter et inclure des templates

L'héritage de template

En voici une partie intéressante ! Plus que la précédente en tout cas. 🤖

Je vous ai fait un *teaser* plus haut : 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 `<?php 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 `<?php 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 - [Sélectionner](#)

```
<!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>
```

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

Code : HTML - [Sélectionner](#)

```
{% 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 ne 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_block %}{% 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 ?

Réponse : 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 : juste après avoir enlevé le nom du contrôleur tout à l'heure, on enlève juste cette fois-ci le nom du *bundle*.

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:form.html.twig` à l'endroit exact pour afficher le formulaire.

On voit bien qu'on ne peut pas faire d'héritage sur le *template* `form.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 :

`{% include "SdzBlogBundle:Blog:form.html.twig" %}`. Ce code inclura le contenu du *template* à l'endroit de la base. Une sorte de copier-coller automatique, en fait ! Exemple tiré de `SdzBlogBundle:Blog:ajouter.html.twig` :

Code : HTML - [Sélectionner](#)

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

{% block body %}

    <h2>Ajouter un article</h2>

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

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

{% endblock %}
```

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 plus tôt, 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.



Encore une fois, vous devez personnaliser tout ça ! Je ne suis pas là pour faire le blog à votre place, juste pour vous guider. Si vous ne pratiquez pas de votre côté en attendant

Encore une fois, vous devez personnaliser tout ça : Je ne suis pas là pour faire le blog à votre place, juste pour vous guider. Et vous ne pratiquez pas de votre côté en ajoutant, supprimant et améliorant tout ce que l'on voit ici, vous serez vite perdu ! Je vous fais confiance, ne faites pas que lire, codez. 😊

Je vais également en profiter pour introduire l'utilisation de CSS dans Symfony2. Cela se fait super bien avec **Assetic**, un *bundle* qui gère les codes CSS et JavaScript. Il est activé par défaut dans Symfony2 (aucune modification à faire dans `AppKernel.php`). Il intègre une balise `{% stylesheets %}` dans Twig qui nous permet de charger tous nos fichiers CSS et JavaScript et permettra par la suite d'appliquer des filtres sur ces fichiers (compression et autre). On ne va pas l'expliquer en détail ici, ça fera l'objet d'un *cookbook*. Regardez simplement comment il est utilisé dans l'exemple et vous saurez l'utiliser de façon basique.

La pratique

Prenons le fichier `app/Resources/views/layout.html.twig` (ce code est ouvertement repris du [cours XHTML de M@teo21](#)):

Code : HTML - [Sélectionner](#)

```
<!DOCTYPE html>
<html>
  <head>
    <title>{% block title %}Sdz{% endblock %}</title>

    {% stylesheets output='css/am.min.css'
      '../app/Resources/public/css/*' %}

    <link rel="stylesheet" href="{% asset_url %}" type="text/css" media="screen" />

    {% endstylesheets %}
  </head>

  <body>
    <div id="en_tete">
      <h1>Ma Première Application Symfony2</h1>
    </div>

    <div id="menu">
      <div class="element_menu">
        <h3>Le blog</h3>
        <ul>
          <li><a href="{% path('sdzblog') %}">Accueil du blog</a></li>
          <li><a href="{% path('sdzblog_ajouter') %}">Ajouter un article</a></li>
        </ul>
      </div>
    </div>

    <div id="corps">

      {% block body %}
      {% endblock %}

    </div>

    <div id="pied_de_page">
      <p>MonSuperBlog propulsé par Symfony2 - Pas de Copyright 2011, aucun
      droit réservé</p>
    </div>
  </body>
</html>
```

J'ai surligné les parties qui contiennent un peu de Twig :

- ligne 4 : création du bloc « title » avec « Sdz » comme contenu par défaut ;
- lignes 6 à 11 : utilisation de `app/Resources/public/css/*.css` grâce à Assetic ;
- lignes 23 et 24 : utilisation de la fonction `{% path %}` pour faire des liens vers d'autres routes ;
- lignes 31 et 32 : création du bloc « body » sans contenu par défaut.

Et voilà, nous avons notre *layout* général ! Voici le CSS qui va bien avec (`app/Resources/public/css/style.css`), histoire d'avoir un truc qui ressemble à quelque chose :

Code : CSS - [Sélectionner](#)

```
/*
Design d'exemple du Site du Zéro
Réalisé par zaz, venom et mateo21
http://www.siteduzero.com
*/

body
{
    width: 760px;
    margin: auto;
    margin-top: 20px;
    margin-bottom: 20px;
    background-color: #CFAFD;
}

/* En-tête */
#en_tete
```

```

{
    width:760px;
    background-color:#FFECFF;
    border:1px dashed blue;
    border-bottom:none;
    margin-bottom:10px;
    text-align:center;
    padding: 0.5em;
}
/* Le menu */
#menu
{
    float:right;
    width:120px;
}
.element_menu
{
    background-color:#FFECFF;
    border:1px dashed blue;
    border-bottom:none;
    padding-bottom:10px;
    margin-bottom:20px;
}
.element_menu h3
{
    color:blue;
    text-align:center;
    font-family:Arial, "Arial Black", "Times New Roman", Times, serif;
}
.element_menu ul
{
    padding:0px;
    padding-left:20px;
    margin:0px;
    margin-bottom:5px;
}
.element_menu a
{
    color:blue;
}
.element_menu a:hover
{
    background-color:purple;
    color:white;
}
/* Le corps de page (la partie principale quoi ^^) : */
#corps
{
    margin-right:140px;
    margin-bottom:20px;
    padding:10px;
    color:#FC00FF;
    background-color:#FFECFF;
    border:1px dashed blue;
    border-bottom:none;
}
#corps h2
{
    height:29px;
    padding-left:30px;
    color:#FC00FF;
    text-align:left;
}
#corps h1
{
    color:#FC00FF;
    font-weight:bold;
    text-align:center;
    font-family:Arial, "Arial Black", "Times New Roman", Times, serif;
}
/* Le pied de page (qui se trouve tout en bas, en général pour les copyrights) : */
#pied_de_page
{
    padding:10px;
    text-align:center;
    color:#3CC500;
    background-color:#FFECFF;
    border:1px dashed blue;
    border-bottom:none;
}

```

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. Une bonne pratique consiste à préfixer le nom des blocs par le nom du *bundle* courant. Regardez le code et vous comprendrez.

La pratique

Extrait du fichier `src/Sdz/BlogBundle/Resources/views/layout.html.twig`:

Code : HTML - [Sélectionner](#)

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

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

{% block body %}

    <h1>Blog</h1>

    {% block sdzblog_body %}
    {% endblock %}

{% endblock %}
```

On a ajouté un `<h1>` dans le bloc « body » puis créé un nouveau bloc qui sera personnalisé par les *templates* finaux. Voyez-vous pourquoi nous n'avons pas appelé ce nouveau bloc « body » ? Car il y aurait eu des soucis avec l'autre bloc « body ». Bref, soyez cohérent dans vos noms.

Les templates finaux

La théorie

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

Blog/liste.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 - [Sélectionner](#)

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

{% block title %}Liste des articles - {{ parent() }}{% endblock %}

{% block sdzblog_body %}

    <h2>Liste des articles</h2>

    <ul>

        {% for article in articles %}

            <li>

                <a href="{{ path('sdzblog_voir', {'slug': article['slug']}) }}">{{ 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 plus haut dans ce chapitre. Modifions le contrôleur en conséquence, juste pour définir la variable `{{ articles }}` (même vide, c'est pour ne pas avoir d'erreur) :

Code : PHP - [Sélectionner](#)

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



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

Vous voulez voir des articles au lieu du message pas très drôle ? Je suis trop bon, voici un tableau d'articles que vous pouvez passer depuis la méthode `render()`, dans `listeAction()` :

Code : PHP - [Sélectionner](#)

```
<?php
public function listeAction()
{
    // ...

    // Les articles :
    $articles = array(
        array('titre' => 'Mon weekend a Phi Phi Island !', 'slug' => 'mon-weekend-a-phi-phi-island', 'auteur' => 'winzou', 'date' => new \Datetime()),
        array('titre' => 'Repetition du National Day de Singapour', 'slug' => 'repetition-du-national-day-de-singapour', 'auteur' => 'winzou', 'date' => new \Datetime()),
        array('titre' => 'Chiffre d'affaire en hausse', 'slug' => 'chiffre-d-affaire-en-hausse', 'auteur' => 'M@teo21', 'date' => new \Datetime())
    );

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

Rechargez la page, et profitez du résultat. 😊

[Blog/voir.html.twig](#)

Je vous laisse le faire tout seul, il ressemble beaucoup à `liste.html.twig`. N'oubliez pas de passer une variable `{{ article }}` depuis le contrôleur.

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

On va faire celui-là car il contient une inclusion. 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 - [Sélectionner](#)

```
{% 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" with {'form': form} %}

    <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>

{% endblock %}
```

Nouveauté : lors de l'inclusion du *template*, nous avons spécifié une variable à transmettre au *template* inclus. En effet, les variables ne sont pas globales et si vous ne passez pas de variables au *template* inclus, celui-ci n'en aura pas de disponible. Ça peut quand même être utile, mais c'est limité. Vous aurez donc souvent à utiliser cette syntaxe du « with ».

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

Quant à `formulaire.html.twig`, on ne sait pas trop le faire encore, mais voici à quoi il ressemblera au début :

Code : HTML - [Sélectionner](#)

```
<form method="post" {{ form_enctype(form) }}>

    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

Deux choses importantes 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*. Il faut seulement lui donner un « with » afin que, ici, la variable du formulaire `{{ form }}` soit disponible ;
- `{{ form_widget(form) }}` affiche tout le formulaire sous forme de tableau. Nous le reverrons plus tard, rassurez-vous, mais c'est une fonctionnalité très utile : après avoir défini le

formulaire côté PHP, pas besoin d'écrire toutes les balises `<input>` côté HTML pour pouvoir tester notre formulaire ! L'appel à `{{ form_widget }}` génère tout et nous pouvons tester directement. Bien sûr, il faudra personnaliser l'affichage par la suite et donc changer ce `{{ form_widget }}`. Nous verrons cela dans la deuxième partie du cours.

Conclusion

Et voilà, nous avons généré 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 !

Voilà, c'est terminé pour ce chapitre ; vous savez afficher avec mise en forme le contenu de votre site.

Ce chapitre clôt la première partie du cours. En effet, vous savez presque tout faire maintenant ! Bon OK, c'est vrai, il vous manque encore des concepts clés. Mais vous maîtrisez pleinement la base et rajouter d'autres concepts par-dessus sera bien plus facile, heureusement.

La deuxième partie du cours va nous permettre d'aller plus loin avec Symfony2. Vous y apprendrez enfin à créer un formulaire, gérer votre base de données et sécuriser votre site !

À bientôt !

Partie 2 : Allons plus loin avec Symfony2

Apprendre à gérer sa 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 dont nous allons parler dans ce chapitre !



Notions d'ORM : fini les requêtes, utilisons des objets

Définition d'ORM : *Object-Relational Mapper*

L'objectif d'un ORM est simple : se charger de la persistance de vos objets (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 Doctrine 2, l'ORM par défaut de Symfony2, pour faire tout cela.

Un rapide exemple pour bien comprendre ? Supposons que vous disposez d'une variable `<?php $utilisateur ?>`, un objet `User`, qui représente l'un de vos utilisateurs. 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. Avec l'ORM, vous n'aurez plus qu'à faire `<?php $orm->save($utilisateur) ?>`, et ce dernier s'occupera de tout ! Bien sûr, ça n'est qu'un exemple, nous verrons les détails pratiques dans la suite de ce chapitre.

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 **O**bjet. 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 accédez à vos attributs via `<?php $utilisateur['pseudo'] /* ou */ $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 */ $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() ?>` ? Ah, ah ! Vous ne pouviez pas faire cela avec votre tableau ! Ici, la méthode `<?php getCommentaires() ?>` déclencherait la bonne requête, récupérerait 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 la persistance à l'ORM s'appelle une **entité** (*entity* en anglais).

Créer une première entité avec Doctrine2

Une entité, c'est juste un objet

Dernière 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 - Sélectionner](#)


```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

class Article
{
    protected $id;

    protected $date;

    protected $titre;

    protected $contenu;

    // les getters
    // les setters
}
```



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 - [Sélectionner](#)

```
<?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->setContenu("C'était vraiment super et on s'est bien amusé.");

    return $this->render('SdzBlogBundle:Article:test.html.twig', array('article' =>
    $article));
}
```

Tout cela avec la vue correspondante qui afficherait l'article passé en argument avec un joli code HTML. Le code est un peu limité car statique, mais l'idée est là et vous voyez comment l'on peut se servir d'une entité.

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 ?

O.K., 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 - [Sélectionner](#)

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */

class Article
{
    /**
     * @ORM\Column(name="id", type="integer")
     */
```

```

    /**
     * @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="contenu", type="text")
     */
    private $contenu;

    // les getters
    // les setters
}

```

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-incrément, etc. Ces informations se nomment les *metadatas* de notre entité. Je ne vais pas épiloguer sur les annotations, elles sont suffisamment claires pour être comprises par tous. 😊

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 *metadatas* d'une entité : en YAML, en XML et en PHP. Vous trouverez plus d'informations sur la définition des *metadatas* dans la [documentation de Doctrine2](#).

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 `php app/console generate:doctrine:entity` et suivez le guide :

1. The Entity shortcut name: : grâce au commentaire juste au-dessus, vous avez compris, il faut rentrer le nom de l'entité sous le format `NomBundle:NomEntité`. Dans notre cas, on entre donc `SdzBlogBundle:Article` ;
2. Configuration format (yaml, xml, php, or annotation) [annotation]: : comme précisé, on va utiliser les annotations qui sont d'ailleurs le format par défaut. Appuyez juste sur la touche Entrée ;
3. 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. 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 » et « contenu ». `Titre` est de type `string` de 255 caractères (pourquoi pas). `Contenu` est par contre de type `text` ;
6. lorsque vous avez fini, appuyez sur la touche Entrée ;
7. Do you want to generate an empty repository class [no]? : oui, on va créer le *repository* associé, c'est très pratique. Entrez donc `yes` ;
8. confirmez la génération, et voilà !

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 ! 🤖

Le repository



Qu'est-ce que ce fichier `ArticleRepository.php` ?

Un *repository* est un objet qui va nous permettre de récupérer nos entités. Pour imaginer, c'est lui qui sait où sont rangées vos entités et comment aller les chercher. Typiquement, pour récupérer l'article d'id 5, on va appeler la méthode `<?php find()` du *repository*, comme cela :

Code : PHP - [Sélectionner](#)

```

<?php
$article_5 = $repository->find(5);

```

Tout simplement !

Bon, c'est super tout cela, mais il est peut-être temps d'enregistrer tout ça dans la base de données !

Manipuler ses entités

Créer la table correspondante dans la base de données

Crée 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, là !

...

Ceux qui m'ont cru, relisez le début du chapitre. 😊 Les autres, venez, on est bien trop fainéants pour ouvrir phpMyAdmin ! Vérifiez tout d'abord si vous avez bien configuré l'accès à votre base de données dans Symfony2, faites-le sinon. Ouvrez le fichier `app/config/parameters.ini` et mettez les bonnes valeurs aux lignes commençant par `database_`.

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. Exécutez donc la commande `php app/console doctrine:schema:create`. Si tout se passe bien, vous aurez droit à un `Database schema created 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-incrément, etc. C'est super !

Générer un CRUD pour manipuler notre entité

Générer

En fait, à ce moment, on voudrait juste tester notre entité en rajoutant des objets et en les affichant, etc. On pourrait le faire depuis phpMyAdmin, mais je trouve ça vraiment trop long. Et je vous promets que vous n'allez pas tarder à penser à la même chose !

Un CRUD, c'est un « *Create, Read, Update and Delete* », et il concerne juste les actions de base que l'on peut faire sur notre objet. C'est un terme très courant qui ne s'applique pas qu'aux entités Doctrine. 😊 Pour ce faire, vous l'aurez compris, le *generator* revient à notre service ! Il va nous générer un contrôleur et les routes correspondantes, et ce contrôleur contiendra toutes les actions pour effectuer les actions CRUD sur notre entité `Article`. C'est parti pour exécuter la commande `php app/console generate:doctrine:crud` :

- The Entity shortcut name : c'est le même nom que l'on a utilisé tout à l'heure, à savoir `SdzBlogBundle:Article` ;
- Do you want to generate the "write" actions [no]? : évidemment que l'on veut, c'est un peu notre but ! Entrez `yes` ;
- Configuration format (yaml, xml, php, or annotation) [annotation]: : on serait tenté de garder les annotations, mais non, ici, Symfony2 va nous générer un contrôleur et des routes, on utilise le format Yaml pour cela, nous. Entrez donc `yaml` ;
- Routes prefix [/article]: : le *generator* ne sait pas gérer le cas où nous avons **déjà** importé les routes de notre *bundle* avec un certain préfixe. Appuyez simplement sur Entrée, nous le réglerons juste après ;
- confirmez la génération, tapez `no` pour mettre à jour le routeur (rappelez-vous, on l'a déjà fait).

Le générateur a donc généré les fichiers `Controller/ArticleController.php` et `Resources/config/routing/article.yml`. Réglons tout de suite l'import des routes de ce fichier. Pour cela on va importer le fichier `article.yml` depuis notre `routing.yml` (celui du bundle, pas celui de `app` !). Je vous laisse le faire tout seul, on a déjà vu la syntaxe. Indice, pas besoin d'utiliser le `@NomDuBundle` puisqu'on est déjà dans le bon répertoire.

Secret (cliquez pour afficher)

Bon, je suis trop bon parce que si vous bloquez ici, on ne peut pas continuer :

Code : Autre - Sélectionner

```
sdzblog_crud:
  resource: "routing/article.yml"
  prefix:   /crud
```

À placer au début du fichier, sinon, notre route `sdzblog_voir` va intercepter des *URL* avant notre CRUD.



Et maintenant, rendez-vous sur http://localhost/Symfony/web/app_dev.php/blog/crud ! Impressionnant, n'est-ce pas ? 😊 Amusez-vous avec, créez des entités, je vous enseignerai la théorie juste après.

Comprendre

Alors, qu'est-ce que nous a généré le générateur ? Côté route, rien de nouveau, vous maîtrisez déjà, mais côté contrôleur, c'est plus intéressant. Allez voir un peu à quoi cela ressemble ! Voici par exemple la méthode `<?php index() ?>` :

Code : PHP - Sélectionner

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

// ...

public function indexAction()
{
    $em = $this->getDoctrine()->getEntityManager();

    $entities = $em->getRepository('SdzBlogBundle:Article')->findAll();

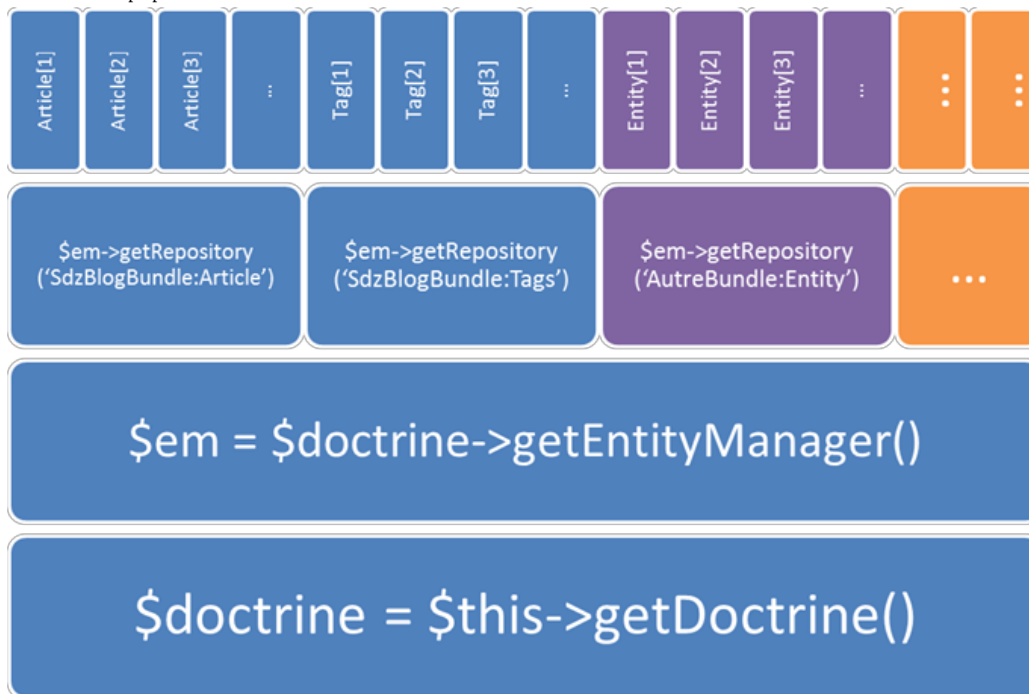
    return $this->render('SdzBlogBundle:Article:index.html.twig', array(
        'entities' => $entities
    ));
}
```

```
}    '''
```

Concernant le service Doctrine, il y a plusieurs choses à savoir. Pour récupérer ce service, vous pouvez soit faire `$this->getDoctrine()`, soit `$this->get('doctrine')`. En effet, Doctrine est un service comme un autre, on peut donc y accéder via la méthode `<?php get() ?>`. Mais comme c'est un service souvent utilisé (et pour cause !), il existe un raccourci via la méthode `<?php getDoctrine() ?>` qui vous permet de bénéficier de l'autocomplétion de votre IDE. Très pratique !

On n'utilisera pas souvent ce service Doctrine brut, ce qui nous intéresse le plus, c'est l'*EntityManager* que l'on récupère avec la méthode `<?php getEntityManager() ?>` depuis le service Doctrine. L'*EntityManager* c'est un peu le chef des entités, il sait **enregistrer** toutes vos entités dans la base de données à la différence de l'*EntityRepository* qui lui ne sait que **récupérer** ses propres entités.

Voici un petit schéma pour illustrer mes propos :



Sachez qu'il peut également y avoir plusieurs *EntityManager* si vous employez plusieurs bases de données, par exemple. Si votre application emploie MySQL et PostgreSQL, vous aurez un [EntityManager](#) par SGBDR.

Utiliser l'EntityManager pour persister nos entités

C'est donc l'*EntityManager* qui sait enregistrer nos entités dans la base de données (on dit **persister**). Visualisons un exemple avec la méthode `<?php createAction() ?>` du contrôleur. Passons sur les détails liés à la gestion du formulaire, le prochain chapitre y est entièrement consacré. Concentrons-nous sur la partie surlignée :

Code : PHP - [Sélectionner](#)

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

public function createAction()
{
    $entity = new Article();
    $request = $this->getRequest();
    $form = $this->createForm(new ArticleType(), $entity);
    $form->bindRequest($request);

    if ($form->isValid()) {
        $em = $this->getDoctrine()->getEntityManager();
        $em->persist($entity);
        $em->flush();

        return $this->redirect($this->generateUrl('admin_article_show', array('id' =>
            $entity->getId())));
    }

    return $this->render('SdzBlogBundle:Article:new.html.twig', array(
        'entity' => $entity,
        'form' => $form->createView()
    ));
}
```

Tout d'abord, à la ligne 4, on crée une entité *Article*. La gestion du formulaire (les lignes suivantes) revient en fait à remplir cette entité avec les valeurs du formulaire. Enfin, on arrive à la partie qui nous intéresse :

- ligne 14 : on charge l'*EntityManager* ;
- ligne 15 : on dit à l'*EntityManager* de persister notre entité. On lui dit en fait : « *Retiens cette entité-là, il faudra que tu l'enregistres dans la base de données.* » ;
- ligne 16 : on dit à l'*EntityManager* d'effectuer les opérations demandées. En effet, à la ligne 11, on dit : « *Tu devras le faire.* », et à la ligne 12, on dit : « *Fais-le maintenant.* ». Attention à bien faire la différence et donc, surtout, à ne pas oublier la méthode `<?php flush() ?>` !

Les transactions

Pourquoi deux méthodes `<?php persist() ?>` et `<?php 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, 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 `<?php persist() ?>` est équivalent à dire : « *Garde cette entité en mémoire, tu l'enregistreras au prochain <?php flush() ?>.* ». Et un `<?php flush() ?>` est équivalent à ceci : « *Ouvre une transaction et enregistre toutes les entités qui t'ont été données depuis le dernier <?php flush() ?>.* ».

L'automatisme

Voilà, vous savez déjà tout sur l'*EntityManager* !

Une précision avant de continuer : la méthode `<?php persist() ?>` traite indépendamment les nouvelles entités des entités déjà persistées. Vous pouvez donc lui passer une entité fraîchement créée comme dans notre exemple ci-dessus, 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 !

Dans le cas d'une nouvelle entité, il générera une requête SQL de type **INSERT INTO** et dans le cas d'une d'entité déjà existante, il générera une requête de type **UPDATE** sur les propriétés qui ont changé (ou rien si rien n'a changé).

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. 😊

Récupérer ses entités avec un EntityRepository

Le rôle de l'EntityRepository

Le rôle de l'*EntityRepository*, comme on l'a déjà vu, est de **récupérer** vos entités. On passera donc très souvent par lui !

Vous avez vu que le générateur a créé une classe *ArticleRepository* qui hérite de `\Doctrine\ORM\EntityRepository`. Dans cette classe, on pourra ajouter nos méthodes personnalisées pour récupérer les entités d'une façon que l'on veut nous.

Mais pour l'instant, étudions déjà les méthodes de l'*EntityRepository* de base de Doctrine, après tout, le nôtre hérite de celui-là, il faut donc le connaître !

Les méthodes de base

Il existe quatre méthodes et une magique. Attaquons d'abord les quatre faciles :

- `<?php find($id) ?>` : on l'a déjà utilisée, `<?php find($id) ?>` retourne tout simplement l'entité correspondant à l'id `<?php $id ?>`. Elle retourne une instance d'*Article* dans le cas de notre *ArticleRepository*, par exemple. On peut difficilement faire plus simple ;
- `<?php findAll() ?>` : *idem*, elle retourne toutes les entités. Elle ne retourne pas un *Array*, mais un *ArrayCollection*. C'est une classe Doctrine qui s'utilise comme un tableau, en fait : vous pouvez faire un `<?php foreach ?>`, des `<?php isset($key) ?>`, etc. ;
- `<?php findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null) ?>` : voici une méthode plus intéressante. Elle est en fait très simple et très intuitive, voici un exemple : `<?php findBy(array('titre' => 'Mon dernier weekend'), array('date' => 'desc'), 5, 0) ?>`. Cet exemple va récupérer toutes les entités ayant comme titre « Mon dernier weekend » en les classant par date décroissante et en en sélectionnant cinq (5) à partir du début (0). Elle retourne un *ArrayCollection* également ;
- `<?php findOneBy(array $criteria) ?>` : même principe que `<?php findBy() ?>`, sauf qu'elle ne retourne qu'une seule entité (elle retourne donc une instance d'*Article* dans notre cas). Comme elle ne retourne qu'une seule entité, les arguments *orderBy*, *limit* et *offset* n'existent pas, évidemment.

Quant à la méthode magique, il s'agit en fait plein de méthodes suivant les propriétés de votre entité. Vous avez :

- `<?php findByXxx($valeur) ?>` : en remplaçant « Xxx » par le nom d'une propriété (dans notre cas, *Titre*, *Date* ou *Contenu*). Elle fonctionne comme `<?php findBy() ?>`, sauf que vous ne pouvez pas préciser les arguments. En fait, `<?php findByXxx($valeur) ?>` fait appel à `<?php findBy(array('Xxx' => $valeur)) ?>` ;
- `<?php findOneByXxx($valeur) ?>` : *idem*, mais elle fonctionne comme `<?php findOneBy() ?>`, c'est-à-dire qu'elle ne retourne qu'une seule entité.

Nos méthodes personnelles

L'*ArticleRepository* généré par Symfony2 sert à pouvoir utiliser des méthodes à nous, qui feront des tâches plus précises que les méthodes de base. Un exemple concret : on souhaite

récupérer tous les articles du blog compris entre deux dates. C'est impossible à faire en employant uniquement les méthodes `<?php find*() ?>` de base ; on va donc créer nos propres méthodes. Voici celle que l'on souhaite créer :

Code : PHP - [Sélectionner](#)

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

class ArticleRepository extends EntityRepository
{
    public function getByDate(\Datetime $date1, \Datetime $date2)
    {
        $queryBuilder = $this->createQueryBuilder('a');
        $queryBuilder->where('a.date BETWEEN :date1 AND :date2')
            ->setParameter('date1', $date1->format('Y-m-d'))
            ->setParameter('date2', $date2->format('Y-m-d'));

        $query = $queryBuilder->getQuery();

        return $query->getResult();
    }
}
```

Ça semble compliqué, mais ça ne l'est pas tant que ça, rassurez-vous.

Le principe est simple : grâce à un objet `QueryBuilder`, on va construire une requête non pas en SQL pur comme on le faisait avant, mais avec des méthodes de ce `QueryBuilder`. Une fois notre requête construite, on va la récupérer grâce à la méthode `<?php getQuery() ?>` du `QueryBuilder`, on obtient donc un objet `Query`. Cet objet, on peut l'exécuter et en récupérer le résultat ; cela se fait grâce à la méthode `<?php getResult() ?>`.

En pratique :

- ligne 7 : on crée un objet `QueryBuilder` grâce à la méthode de l'`EntityRepository` `<?php createQueryBuilder($alias) ?>` qui crée un objet `QueryBuilder` en effectuant déjà la requête **SELECT** article **FROM** Sdz\BlogBundle\Entity\Article **AS** \$alias. On n'a plus qu'à faire le reste de la requête, ou même à changer le début : il est tout à fait possible de remplacer le **SELECT** par un **UPDATE**, par exemple ;
- ligne 8 : on rajoute un **WHERE** à notre requête, avec la méthode `<?php where() ?>`, tout simplement ; ici, un simple **BETWEEN** sur notre date ;
- lignes 9 et 10 : on attache nos paramètres ;
- ligne 12 : on récupère l'objet `Query` depuis notre `QueryBuilder`. Maintenant, on ne peut plus modifier la requête, mais elle n'est pas encore exécutée ;
- ligne 14 : on exécute la requête, récupère les résultats, puis on les retourne.

Et voilà ! Pour vérifier le bon comportement de notre méthode fraîchement créée, on va modifier le contrôleur, ligne 24. Dans `<?php indexAction() ?>`, remplacez :

Code : PHP - [Sélectionner](#)

```
<?php
// src/Sdz/BlogBundle/Controller/ArticleController.php
// ligne 24 :
$entities = $em->getRepository('SdzBlogBundle:Article')->findAll();
```

Par :

Code : PHP - [Sélectionner](#)

```
<?php
// src/Sdz/BlogBundle/Controller/ArticleController.php
// ligne 24 :
$entities = $em->getRepository('SdzBlogBundle:Article')->getByDate(new \Datetime('2010-01-01'),
    new \Datetime('2010-12-31'));
```

On a pris tous les articles de l'année 2010, vérifiez que cela fonctionne bien avec les entités que vous avez. 😊

Pour savoir utiliser les objets `QueryBuilder` et `Query` de fond en comble, je vous invite à toujours avoir sous la main [la page de documentation de Doctrine à ce sujet](#). On ne va pas le traiter ici, donc lisez vraiment cette page, vous en aurez besoin pour construire vos requêtes. Et vous avez la base, donc c'est super facile à comprendre.

Pour vous aider, voici quelques exemples. Pour les comprendre, ouvrez un onglet avec la documentation à côté, et décortiquez les requêtes. Notez que ce sont des exemples volontairement sans rapport avec notre blog, mais vous pouvez cependant les transposer facilement, car ce sont des cas classiques. Vous pourrez vous entraîner à la fin de ce chapitre, lorsque nous aurons une base de données fonctionnelle ; patience. 😊

Code : PHP - [Sélectionner](#)

```
<?php
// src/Sdz/BlogBundle/Entity/ArticleRepository.php

class ArticleRepository extends EntityRepository
{
    // Fonction du repository Membre pour supprimer tous les membres d'un Groupe.
    public function deleteByGroup(Groupe $groupe)
    {
    }
```

```

        $qb = $this->_em->createQueryBuilder()
            ->delete($this->_entityName, 'm') // $this->_entityName correspond au nom de
            l'entité gérée par ce repository. Ici, on aurait pu mettre directement «
            Namespace\Bundle\Entity\Membre ».
            ->where('m.groupe = :groupe')
            ->setParameter('groupe', $groupe);

        return $qb->getQuery()->execute(); // execute() permet d'exécuter la requête sans
        récupérer son résultat (car pour un DELETE, il n'y a pas de résultat).
    }

    // Fonction du repository Account pour sélectionner tous les « accounts » qui ont un wrap_id
    compris dans le tableau en argument.
    public function getByWrap(array $list_wrap_ids)
    {
        $qb = $this->createQueryBuilder('a');

        // Notez l'utilisation de l'expr(), utile dans certains cas, cf. la doc.
        $qb ->where($qb->expr()->in('a.wrap_id', $list_wrap_ids));

        return $qb->getQuery()->getResult();
    }

    // Fonction du repository Membre pour compter le nombre total de membres.
    public function getCount()
    {
        return $this
            ->createQueryBuilder('m')
            ->select('COUNT(m)') // Le createQueryBuilder() définit déjà un SELECT, mais
            on le remplace avec le nôtre.
            ->getQuery()
            ->getSingleScalarResult(); // getSingleScalarResult() permet de récupérer
            directement le COUNT.
            // Single = on n'attend qu'une seule ligne de résultat
            // Scalar = cette unique ligne n'a qu'une seule colonne,
            on récupère directement sa valeur
    }
}

```

Établir des relations entre nos entités

Les différents types de relations

Stocker vos objets indépendamment les uns des autres, c'est très facile, mais ça n'est pas très intéressant. On a besoin de lier des commentaires à un article, des tags à un article également, etc. Bref, d'établir des relations entre nos entités.

Les relations sont très bien intégrées dans nos entités. Ainsi, pour décrire les tags de nos articles par exemple, on va simplement rajouter une propriété tags à notre entité `Article`. Doctrine s'occupera de tout dans les coulisses.

Il existe plusieurs types de relations qui sont assez intuitifs, car ils correspondent à des cas concrets.

0. Notion de propriétaire et d'inverse

Cette notion un peu abstraite est assez 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.**

Prenons un exemple simple, toujours les commentaires d'un article de blog. Vous disposez de la table « commentaire » et de la table « article », très bien. 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 ».

On ne va traiter ici que les relations 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() ?>`). C'est une limitation qui peut être contournée en utilisant les relations bidirectionnelles, expliquées dans la documentation sur les associations : [ce chapitre](#) d'abord, puis [celui-ci](#) ensuite. Attention, cela ne nous empêchera pas de récupérer les commentaires d'un article ! On utilisera juste une autre méthode, via l'*EntityRepository*.

1. Relation One-To-One

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. Ainsi, dans une application avec des clients et leur adresse, la relation entre `Client` et `Adresse` est une relation *One-To-One*. En effet, un client correspond à une seule adresse, et une adresse correspond à un seul client.

Pour établir cette relation dans votre entité, la syntaxe est la suivante :

Code : PHP - Sélectionner

```

<?php
/**
 * @ORM\Entity
 */
class Client

```

```

{
    /**
     * @ORM\OneToOne(targetEntity="Namespace\Bundle\Entity\Adresse")
     */
    private $adresse;

    // ...
}

/**
 * @ORM\Entity
 */
class Adresse
{
    // Nul besoin de rajouter une propriété, ici.

    // ...
}

```

C'est simple, non ? On utilise l'annotation *OneToOne* pour définir la relation. *TargetEntity* définit simplement l'entité à l'autre bout de la relation.

Vous devez voir que j'ai défini *Client* comme l'entité propriétaire de la relation (car c'est lui qui contient la référence). *Adresse* est donc l'inverse.

Pour utiliser cette relation, c'est très simple. Dans le fichier de l'entité :

Code : PHP - [Sélectionner](#)

```

<?php
// Dans le fichier de l'entité.

/**
 * @ORM\Entity
 */
class Client
{
    /**
     * @ORM\OneToOne(targetEntity="Namespace\Bundle\Entity\Adresse")
     */
    private $adresse;

    // On définit le getter et le setter associé.
    public function getAdresse()
    {
        return $this->adresse;
    }

    // Ici, on force le type de l'argument à être une instance de notre entité Adresse.
    public function setAdresse(Namespace\Bundle\Entity\Adresse $adresse)
    {
        $this->adresse = $adresse;
    }
}

```

Et dans le fichier du contrôleur :

Code : PHP - [Sélectionner](#)

```

<?php

// Dans le contrôleur, pour ajouter un nouveau client et son adresse :
public function creerClientAction()
{
    $client = new Client;
    // $client->setNom() etc

    $adresse = new Adresse;
    // $adresse->setRue() etc

    // On lie l'adresse au client
    $client->setAdresse($adresse);

    // On récupère l'EntityManager
    $em = $this->getDoctrine()->getEntityManager();

    // On persiste nos deux entités
    $em->persist($client);
    $em->persist($adresse);

    // On déclenche l'enregistrement
    $em->flush();

    return new Response('OK');
}

// Dans le contrôleur toujours, mais pour modifier l'adresse déjà existante

```



```
// d'un client déjà existant :
public function modifierAdresseAction($id_client)
{
    $em = $this->getDoctrine()->getEntityManager();

    // On récupère le client.
    $client = $em->getRepository('SdzBlogBundle:Client')->find($id_client);

    // On modifie le numéro de l'adresse.
    $client->getAdresse()->setNumero(35);

    // On persiste notre client.
    $em->persist($client);

    // Pas besoin de persister l'adresse ici, car elle est déjà gérée par
    // le client qui en est le propriétaire.

    // On déclenche la modification.
    $em->flush();

    return new Response('OK');
}
```

Le code parle de lui-même : gérer une relation est vraiment aisé avec Doctrine !

2. Relation Many-To-One

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 un objet de l'entité 1 d'avoir une relation avec plusieurs objets de l'entité 2. Restons avec notre exemple des clients et leur adresse, et imaginons maintenant que notre application permette à nos clients d'enregistrer plusieurs adresses. Nous avons ainsi plusieurs adresses (*Many*) à lier (*To*) à un seul client (*One*).

Pour établir cette relation dans votre entité, la syntaxe est la suivante :

Code : PHP - [Sélectionner](#)

```
<?php
/**
 * @ORM\Entity
 */
class Client
{
    // Nul besoin de rajouter de propriété, ici.

    // ...
}

/**
 * @ORM\Entity
 */
class Adresse
{
    /**
     * @ORM\ManyToOne(targetEntity="Namespace\Bundle\Entity\Client")
     */
    private $client;

    // ...
}
```

L'annotation à utiliser est *ManyToOne*, intuitif.

Première remarque : l'entité propriétaire a changé, c'est maintenant *Adresse*. Pourquoi ? Parce que rappelez-vous, le propriétaire est celui qui contient la colonne référence. Ici, on aura bien une colonne « client_id » dans la table « adresse ». 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 adresses pour un seul client, le *Many* correspond aux adresses, donc l'entité *Adresse* est la propriétaire.

La méthode pour gérer les multiples adresses est un peu différente, voyez dans cet exemple :

Code : PHP - [Sélectionner](#)

```
<?php
// Dans le fichier de l'entité.

/**
 * @ORM\Entity
 */
class Client
{
    // On a vu que l'on ne peut pas récupérer les adresses depuis le côté inverse de la
    // relation.
    // Donc il n'y a pas de propriété ni de getter/setter.
}

/**
```

```

* @ORM\Entity
*/
class Adresse
{
    /**
     * @ORM\ManyToOne(targetEntity="Namespace\Bundle\Entity\Client")
     */
    private $client;

    // On définit le getter et le setter associé.
    public function getClient()
    {
        return $this->client;
    }

    // Ici, on force le type de l'argument à être une instance de notre entité Client.
    public function setClient(Namespace\Bundle\Entity\Client $client)
    {
        $this->client = $client;
    }
}

```

Et dans le contrôleur :

Code : PHP - [Sélectionner](#)

```

<?php
// Dans le contrôleur, pour ajouter un nouveau client et ses adresses :

use Symfony\Component\Security\Core\Exception\AccessDeniedException;

public function creerClientAction()
{
    $client = new Client;
    // $client->setNom() etc

    $adresse1 = new Adresse;
    // $adresse1->setRue() etc
    $adresse2 = new Adresse;
    // $adresse2->setRue() etc

    // On lie les adresses au client.
    $adresse1->setClient($client);
    $adresse2->setClient($client);

    // On récupère l'EntityManager.
    $em = $this->getDoctrine()->getEntityManager();

    // On persiste nos trois entités.
    $em->persist($client);
    $em->persist($adresse1);
    $em->persist($adresse2);

    // On déclenche l'enregistrement.
    $em->flush();

    return new Response('OK');
}

// Dans le contrôleur toujours, mais pour modifier une adresse déjà existante :
public function modifierAdresseAction($id_adresse)
{
    $em = $this->getDoctrine()->getEntityManager();

    // On récupère l'adresse.
    $adresse = $em->getRepository('SdzBlogBundle:Adresse')->find($id_adresse);

    // On vérifie que l'adresse correspond au client en session (par exemple).
    if($adresse->getClient()->getId() != $this->get('session')->get('client_id'))
    {
        throw new AccessDeniedException("Vous n'avez pas le droit de modifier cette adresse.");
    }

    // On modifie le numéro de l'adresse.
    $adresse->setNumero(35);

    // On persiste notre adresse.
    $em->persist($adresse);

    // On ne passe pas du tout par le client ici, en fait.

    // On déclenche la modification.
    $em->flush();

    return new Response('OK');
}

```

3. Relation Many-To-Many

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 ! O.K., prenons l'exemple cette fois-ci des membres d'un site, divisés en groupes. Un membre peut appartenir à plusieurs groupes. À l'inverse, un groupe peut contenir plusieurs membres. On a donc une relation *Many-To-Many* entre *Membre* et *Groupe*.

Cette relation est particulière dans le sens où Doctrine va devoir créer une table intermédiaire. En effet, comment faites-vous pour faire ce genre de relation ? Vous avez une table « membre », une autre table « groupe », mais vous avez surtout besoin d'une table « membre_groupe » qui fait la liaison entre les deux ! Cette table de liaison ne contient que deux colonnes : « membre_id » et « groupe_id ». Cette table intermédiaire, vous ne la connaissez pas : elle n'apparaît pas dans nos entités, et c'est Doctrine qui la crée et qui la gère tout seul !

Pour établir cette relation dans vos entités, la syntaxe est la suivante :

Code : PHP - [Sélectionner](#)

```
<?php
/**
 * @ORM\Entity
 */
class Membre
{
    /**
     * @ORM\ManyToMany(targetEntity="Namespace\Bundle\Entity\Groupe")
     */
    private $groupes;

    // ...

    /**
     * @ORM\Entity
     */
    class Groupe
    {
        // Nul besoin d'ajouter une propriété, ici.

        // ...
    }
}
```

J'ai mis *Membre* comme propriétaire de la relation. C'est un choix que vous pouvez faire comme bon vous semble, ici. Mais bon, récupérer les groupes d'un membre se fera assez souvent, alors que récupérer les membres d'un groupe, moins. Et puis, pour récupérer les membres d'un groupe, on aura sûrement besoin de personnaliser la requête, donc on le fera de toute façon depuis le *GroupeRepository*.

La méthode pour gérer les groupes multiples est un peu différente, car on a pour la première fois une propriété (ici `<?php $groupes ?>`) qui contient **une liste d'objets**. Voyez l'application concrète dans cet exemple :

Code : PHP - [Sélectionner](#)

```
<?php
// Dans le fichier de l'entité.

/**
 * @ORM\Entity
 */
class Membre
{
    /**
     * @ORM\ManyToMany(targetEntity="Namespace\Bundle\Entity\Groupe")
     */
    private $groupes;

    // Comme la propriété groupes doit être un ArrayCollection, souvenez-vous ;
    // on doit la définir dans un constructeur.
    public function __construct()
    {
        $this->groupes = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // On est dans le côté propriétaire, on définit le getter.
    // Notez le « s » à getGroupes, on récupère une liste de groupes, ici !
    public function getGroupes()
    {
        return $this->groupes;
    }

    // Le setter. Attention, on n'a pas un setGroupes() mais un addGroup() !
    public function addGroupe(\Namespace\Bundle\Entity\Groupe $groupe)
    {
        // On traite vraiment notre ArrayCollection comme un tableau.
        $this->groupes[] = $groupe;
    }
}
```

```

/**
 * @ORM\Entity
 */
class Groupe
{
    // Rien de spécial ici, vous pouvez passer votre chemin.
}

```

Et dans le contrôleur :

Code : PHP - Sélectionner

```

<?php
// Dans le contrôleur, pour ajouter un nouveau membre à plusieurs groupes existants :

public function ajouterMembreAction()
{
    $membre = new Membre;
    // $membre->setPseudo() etc

    // On récupère l'EntityManager.
    $em = $this->getDoctrine()->getEntityManager();

    // On récupère les groupes par défaut à l'inscription d'un membre.
    // Cette méthode n'existe pas, il faudrait la créer, bien sûr.
    $groupes = $em->getRepository('SdzBlogBundle:Groupe')->getParDefaut();

    // Pour chaque groupe, on y ajoute le nouveau membre.
    foreach($groupes as $groupe)
    {
        $membre->addGroup($groupe);
    }

    // On persiste juste le membre.
    // En effet, les groupes existent déjà et on ne les modifie pas, donc pas besoin de les
    persister.
    // Et la relation entre membre et groupe, c'est le membre qui la gère,
    // donc persister le membre persistera la relation.
    $em->persist($membre);

    // On déclenche l'enregistrement.
    $em->flush();

    return new Response('OK');
}

// Dans le contrôleur toujours, mais pour enlever d'un groupe un membre :
public function enleverGroupeAction($groupe)
{
    $em = $this->getDoctrine()->getEntityManager();

    // On récupère le membre, bien sûr on suppose ici que son id est stocké en session.
    $membre_id = $this->get('session')->get('membre_id');
    $membre = $em->getRepository('SdzBlogBundle:Membre')->find($membre_id);

    // On enlève le groupe.
    // Ici, on fait appel à la méthode remove() de l'ArrayCollection groupes.
    $membre->getGroupes()->removeElement($groupe);

    // On a modifié la relation membre-groupe, or c'est le membre qui en est le propriétaire.
    // Donc on persiste le membre pour persister la relation.
    $em->persist($membre);

    // On déclenche la modification.
    $em->flush();

    return new Response('OK');
}

```

Application : les entités de notre blog

Plan d'attaque

On a déjà bien traité l'entité `Article` au début de ce chapitre. J'ai volontairement omis la propriété « pseudo », qui est le pseudo de celui qui a écrit l'article. Souvenez-vous, pour commencer, on n'a pas d'entité `Utilisateur`, on va donc écrire le pseudo de l'auteur en dur dans les articles. Ajoutez donc (à la main, vous ne pouvez plus utiliser le générateur) une propriété « pseudo », de type `string`, sans oublier le `getter` et le `setter`.

Il manque l'entité `Tag`. On va la faire très simple : un tag n'aura qu'une propriété utile, son nom. Créez l'entité indépendamment de la relation qu'elle aura avec `Article`, vous pouvez le faire avec le générateur.

Maintenant, établissez la relation entre `Article` et `Tag`. Je vous laisse réfléchir pour identifier le type de relation qui est le propriétaire. Définissez la propriété de relation et le

getter/setter qui va bien avec.

N'oubliez pas de mettre à jour la base de données avec la commande `php app/console doctrine:schema:update --force`. Attention, la commande `doctrine:schema:create` nous a servi à **créer** la base de données, mais maintenant, nous voulons la **mettre à jour**. On utilisera donc *update* à l'avenir.

Puis, allez remplir des données dans la table avec phpMyAdmin, le générateur ne pouvant pas vous générer de quoi ajouter des tags à un article. Rappelez-vous, il n'est qu'une base pour notre code, pas la solution toute faite. 😊

Enfin, ajoutez une méthode dans l'*ArticleRepository* pour récupérer tous les articles qui correspondent à une liste de tags. La définition de la méthode est donc `<?php getAvecTags(array $nom_tags) ?>`, que l'on pourra utiliser comme cela par exemple : `<?php $articleRepository->getAvecTags(array('sdz', 'weekend')) ?>`.

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 handicapera par la suite !

Le code

Article.php :

Code : PHP - [Sélectionner](#)

```
<?php
// src/Sdz/BlogBundle/Entity/Article.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\ArrayCollection;

use Sdz\BlogBundle\Entity\Tag;

/**
 * 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 Datetime $date
     */
    /**
     * @ORM\Column(name="date", type="date")
     */
    private $date;

    /**
     * @var string $titre
     */
    /**
     * @ORM\Column(name="titre", type="string", length=255)
     */
    private $titre;

    /**
     * @var text $contenu
     */
    /**
     * @ORM\Column(name="contenu", type="text")
     */
    private $contenu;

    /**
     * @var string $pseudo
     */
    /**
     * @ORM\Column(name="pseudo", type="string", length=255)
     */
    private $pseudo;

    /**
     * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Tag")
     */
```

```

    private $tags;

    public function __construct()
    {
        $this->tags = new ArrayCollection();
    }

    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set date
     *
     * @param \Datetime $date
     */
    public function setDate(\Datetime $date)
    {
        $this->date = $date;
    }

    /**
     * Get date
     *
     * @return \Datetime
     */
    public function getDate()
    {
        return $this->date;
    }

    /**
     * Set titre
     *
     * @param string $titre
     */
    public function setTitre($titre)
    {
        $this->titre = $titre;
    }

    /**
     * Get titre
     *
     * @return string
     */
    public function getTitre()
    {
        return $this->titre;
    }

    /**
     * Set contenu
     *
     * @param text $contenu
     */
    public function setContenu($contenu)
    {
        $this->contenu = $contenu;
    }

    /**
     * Get contenu
     *
     * @return text
     */
    public function getContenu()
    {
        return $this->contenu;
    }

    /**
     * Set pseudo
     *
     * @param string $pseudo
     */
    public function setPseudo($pseudo)
    {
        $this->pseudo = $pseudo;
    }

    /**

```

```

    /**
     * Get pseudo
     *
     * @return string
     */
    public function getPseudo()
    {
        return $this->pseudo;
    }

    /**
     * Add tag
     *
     * @param Tag $tag
     */
    public function addTag(Tag $tag)
    {
        $this->tags[] = $tag;
    }

    /**
     * Get tags
     *
     * @return ArrayCollection
     */
    public function getTags()
    {
        return $this->tags;
    }
}

```

Notez que j'ai forcé la variable « date » à être une instance de `\DateTime`. En effet, c'est ce que Doctrine utilise, c'est ce qu'il va nous retourner lorsque l'on fait des requêtes. Donc autant travailler avec un objet `\DateTime` partout dans notre code, et oublier les chaînes de caractères `<?php 'dd/mm/YY' ?>`, ça fait tellement années 90 ! Et n'ayez crainte, depuis un *template* Twig, vous pourrez faire `{{ article.date|date('dd/mm/YY') }}` pour transformer votre `\DateTime` en quelque chose de lisible pour vos visiteurs.

Pour plus d'informations sur la classe `DateTime` de PHP, je vous invite à lire [sa documentation](#).

Tag.php :

Code : PHP - [Sélectionner](#)

```

<?php
// src/Sdz/BlogBundle/Entity/Tag.php

namespace Sdz\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Sdz\BlogBundle\Entity\Tag
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Sdz\BlogBundle\Entity\TagRepository")
 */
class Tag
{
    /**
     * @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 $nom
*/
public function setNom($nom)
{
    $this->nom = $nom;
}

/**
* Get nom
* @return string
*/
public function getNom()
{
    return $this->nom;
}
}

```

ArticleRepository.php :

Code : PHP - [Sélectionner](#)

```

<?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 getAvecTags(array $tags)
    {
        $qb = $this->createQueryBuilder('a');

        // On fait une jointure sur la table des tags, avec pour alias « t ».
        $qb ->join('a.tags', 't')
            ->where($qb->expr()->in('t.nom', $tags)); // Puis on filtre sur le nom des tags.

        // Enfin, on retourne le résultat.
        return $qb->getQuery()->getResult();
    }
}

```



Que faire avec ce que retourne cette fonction ?

Comme je l'ai dit plus haut, cette fonction va retourner un `ArrayCollection` d'`Article`. Qu'est-ce que l'on veut en faire ? Les afficher. Donc la première chose à faire est de passer cet `ArrayCollection` à Twig. Ensuite, dans Twig, vous faites un simple `{% for %}` pour afficher les articles. En fait, c'est simple, regardez comment fonctionne la méthode `indexAction()` du contrôleur `Article`, puis son `template` en `Article:index.html.twig`. 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 tutoriel : entraînez-vous ! Amusez-vous à faire des requêtes dans tous les sens dans l'`ArticleRepository` ou même dans `TagRepository`. Jouez avec la relation entre les deux entités, créez-en d'autres. Bref, ça ne viendra pas tout seul, il va falloir travailler un peu de votre côté. 😊

Vous savez maintenant comment vous servir de l'outil qui va gérer la base de données pour vous, Doctrine2. Grâce à lui, vous pouvez maintenant aller bien plus loin dans votre application. Je vous déconseille d'attendre les prochains chapitres avant de vous entraîner à manipuler les entités, les `EntityRepository` et l'`EntityManager`.

Pour avoir sous les yeux tout ce qu'il faut savoir sur Doctrine2, je vous propose de télécharger la cheat sheet de Elao, une agence web qui fait du Symfony2 : <http://www.elao.org/wp-content/uploads/.../sheet-all.pdf>

Prochain chapitre au programme : les formulaires, pour remplir dynamiquement vos entités !

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.

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**. En effet, que ceux qui pensent pouvoir réutiliser facilement leur formulaire de création d'un article de blog dans une autre page lèvent la main... Personne ? C'est bien ce que je pensais. 🙄

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**.

Reprenons cette définition.

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.

Prenons un exemple pour illustrer cela : un formulaire de recherche. *A priori*, nous n'avons pas d'entité `Recherche` car cela n'a pas de sens (sauf si vous souhaitez enregistrer en base de données toutes les recherches effectuées, pourquoi pas), et pourtant, on a bien besoin d'un formulaire de recherche ! Il suffit donc de créer un simple objet `Recherche`, composé d'un seul attribut « requête », par exemple. Cet objet est suffisant pour construire notre formulaire, et ce n'est pas une entité Doctrine2.

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) :

Secret ([cliquez pour afficher](#))

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Entity;

use Sdz\BlogBundle\Entity\Tag;

class Article
{
    private $id;
    private $date;
    private $titre;
    private $contenu;
    private $pseudo;
    private $tags;

    public function __construct()
    {
        $this->tags = new ArrayCollection();
    }

    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 setContenu($contenu)
    {
        $this->contenu = $contenu;
    }
    public function getContenu()
    {
        return $this->contenu;
    }

    public function getPseudo()
    {
        return $this->pseudo;
    }
    public function setPseudo($pseudo)
    {
        $this->pseudo = $pseudo;
    }

    public function addTag(Tag $tag)
    {
        $this->tags[] = $tag;
    }
    public function getTags()
    {
        return $this->tags;
    }
}

```



Rappel : la convention pour le nom des *getters/setters* est importante : lorsque l'on parlera du champ « nom », le composant *Form* utilisera l'objet via les méthodes `<?php setPseudo() ?>` et `<?php getPseudo() ?>` (comme le faisait Doctrine2 de son côté). Donc si vous aviez eu `<?php set_pseudo() ?>` ou `<?php recuperer_pseudo() ?>`, 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 setPseudo('winzou') ?>` et `<?php 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 : faire une recherche dans le cas d'un objet `Recherche`, 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***.

Pour faire des tests, placez-vous dans l'action `<?php ajouterAction() ?>` de notre contrôleur `Blog`. Modifiez-la comme suit :

Code : PHP - [Sélectionner](#)

```

<?php
use Sdz\BlogBundle\Entity\Article;

// ...

public function ajouterAction()
{
    // On crée notre objet Article.
    $article = new Article();

    // On crée le FormBuilder grâce à la méthode du contrôleur.
    $formBuilder = $this->createFormBuilder($article);

```

```

// On ajoute les champs à notre formulaire.
$formBuilder
    ->add('date', 'date')
    ->add('titre', 'text')
    ->add('contenu', 'textarea')
    ->add('pseudo', 'text'); // Pour l'instant, pas de tags, on les gèrera 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(),
));
}

```

Avec ce premier code, il n'est pas encore question de gérer la soumission du formulaire. Ce code ne fait qu'afficher le formulaire. Il est d'ailleurs déjà fonctionnel, rendez-vous à l'adresse [http://localhost/Symfony/web/app_dev.php\[...\]outer/article](http://localhost/Symfony/web/app_dev.php[...]outer/article). Il est assez explicite en lui-même, mais expliquons un peu la démarche.

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 `<?php $article ?>`.* », puis : « *Ajoute les champs « date », « titre », « contenu » et « pseudo ».* » Et enfin : « *Maintenant, donne-moi le formulaire construit avec tout ce que je t'ai dit avant.* »

Enfin, c'est avec cet objet `<?php $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 createView() ?>` qui permet à la vue d'afficher dès maintenant un formulaire basique avec un tableau. Sans avoir à écrire une seule ligne de code HTML, vous avez déjà pu voir le résultat !



Pourquoi n'a-t-on pas rajouté de champ « id » ? C'est pourtant une propriété à part entière de notre objet, non ?

Prenons dès maintenant un bon réflexe : le formulaire que vous construisez autour d'un objet ne remplit pas forcément **toutes** les propriétés de cet objet ! Rappelez-vous du chapitre précédent : Doctrine2 va remplir automatiquement l'id grâce à l'auto-incrémentation. Donc dans notre formulaire, on n'a pas besoin d'hydrater cette propriété. Elle reste vide jusqu'à ce que Doctrine2 persiste l'objet, en remplissant la propriété avec la bonne valeur.

À l'inverse, on verra que tous les champs d'un formulaire ne sont pas forcément directement liés à l'objet. Regardez par exemple le champ « date » généré : il comprend trois sous-champs qui ne sont aucunement liés à une propriété d'*Article*.

Ajouter des champs

Vous pouvez le voir, ça se fait assez facilement avec la méthode `<?php add() ?>` du *FormBuilder*. Les arguments sont les suivants :

- 1. le nom du champ ;
- 2. le type du champ : [liste complète dans la documentation](#) ;
- 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 ! 🧙



La liste complète des types de champ se trouve [dans la doc](#). Je vous ordonne d'y aller, elle regorge de types très intéressants !

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 `<?php bindRequest() ?>` 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 leur exactitude. Il faut valider notre objet. Cela se fait via la méthode `<?php 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. Et d'ailleurs, pour récupérer cet objet, comment faire ? La méthode `<?php getData() ?>` du formulaire est là pour vous.

Vous êtes un peu perdu ? C'est parce que vous manquez de code. Voici comment faire tout ce que l'on vient de dire :

Code : PHP - Sélectionner

```

<?php
use Sdz\BlogBundle\Entity\Article;

// ...

public function ajouterAction()
{
    // J'ai raccourci cette partie, car plus rapide à écrire !
    $form = $this->createFormBuilder(new Article)
        ->add('date', 'date')
        ->add('titre', 'text')
        ->add('contenu', 'textarea')
        ->add('pseudo', 'text')
        ->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.
        $form->bindRequest($request);

        // On vérifie que les valeurs entrées sont correctes.
        // (Nous verrons la validation des objets en détail plus bas dans ce chapitre.)
        if( $form->isValid() )
        {
            // On récupère notre objet.
            $article = $form->getData();

            // On l'enregistre dans la base de données.
            $em = $this->getDoctrine()->getEntityManager();
            $em->persist($article);
            $em->flush();

            // On redirige vers la page d'accueil, par exemple.
            return $this->redirect($this->generateUrl('sdzblog'));
        }
    }

    // À ce stade :
    // - soit la requête est de type « GET », donc le visiteur vient d'arriver 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, par exemple. Vous voyez que ce formulaire gère déjà très bien les erreurs, il n'enregistre l'article que lorsque tout va bien.

Externaliser la définition du formulaire

Vous savez enfin créer un formulaire. Ce n'était pas très compliqué, nous l'avons rapidement créé 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**. On va mettre tous nos `xxxType.php` dans le répertoire `Form` du *bundle*. En fait, allez voir, ce répertoire existe déjà, et le `ArticleType` également ! En effet, Symfony2 l'avait généré lorsque l'on avait créé le CRUD. Ajoutez juste le champ « pseudo » que l'on avait rajouté *a posteriori* et que le générateur n'avait donc pas intégré.

Et prenez aussi dès maintenant un bon réflexe, que le générateur n'a malheureusement pas respecté. En effet, rappelez-vous, un formulaire se construit autour d'un objet, et avec la technique du `ArticleType`, nous pouvons économiser du temps en définissant directement dans `ArticleType` quel est l'objet sous-jacent. Ainsi, nous n'aurons pas besoin de créer une instance et de la lui donner en argument. Cela se fait très simplement en ajoutant la méthode `<?php getDefaultOptions() ?>` dans notre `ArticleType` :

Code : PHP - [Sélectionner](#)

```

<?php
class ArticleType extends AbstractType
{
    // ...

    public function getDefaultOptions(array $options)

```

```

    {
        return array(
            'data_class' => 'Sdz\BlogBundle\Entity\Article',
        );
    }
}

```

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 !

Le contrôleur épuré

Avec cet ArticleType, la construction du formulaire côté contrôleur se réduit à cette ligne :

Code : PHP - [Sélectionner](#)

```

<?php
// ...
use Sdz\BlogBundle\Form\ArticleType;
// ...
$form = $this->createForm(new ArticleType());
// ...

```

Au final, en utilisant cette externalisation et en supprimant les commentaires, voilà à quoi ressemble la gestion d'un formulaire dans Symfony2 :

Code : PHP - [Sélectionner](#)

```

<?php
use Sdz\BlogBundle\Entity\Article;
use Sdz\BlogBundle\Form\ArticleType;

// ...

public function ajouterAction()
{
    $form = $this->createForm(new ArticleType());

    $request = $this->get('request');
    if( $request->getMethod() == 'POST' )
    {
        $form->bindRequest($request);
        if( $form->isValid() )
        {
            $article = $form->getData();

            $em = $this->getDoctrine()->getEntityManager();
            $em->persist($article);
            $em->flush();

            return $this->redirect( $this->generateUrl('sdzblog' ) );
        }
    }

    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 toute autre action dont votre site Internet aura besoin.



N'hésitez pas à tester votre formulaire en ajoutant des articles !

Gérer les valeurs par défaut du formulaire

L'un des besoins courant 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 faut juste le savoir :

Code : PHP - [Sélectionner](#)

```
<?php
// Création d'une instance d'Article pour y mettre des valeurs par défaut.
$article = new Article();
$article->setDate(new \DateTime()); // Ici, on pré-rempli avec la date
d'aujourd'hui, par exemple.
$form = $this->createForm(new ArticleType(), $article); // Notez le second argument.

// Ou

// Édition d'un article déjà existant.
$article = $this->getDoctrine()->getRepository('Sdz\BlogBundle\Entity\Article')->find($id);
$form = $this->createForm(new ArticleType(), $article);
```

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 patte 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 - [Sélectionner](#)

```
<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 », « pseudo » et « tags »,
mais aussi le champ CSRF (géré automatiquement par Symfony !)
et tous les champs cachés (type « hidden »). -->
{{ form_rest(form) }}
```

[Plus d'informations sur le Cross Site Request Forgeries \(CSRF\).](#)

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 ? Créer un nouveau type de champ !

Je ne vais pas décrire la démarche ici, mais sachez que ça existe et que [c'est bien ce sera bientôt documenté](#).

Les formulaires imbriqués

Intérêts de l'imbrication

En fait, pourquoi imbriquer des formulaires ?

C'est souvent le cas lorsque vous avez des relations entre vos objets : vous souhaitez ajouter un objet, mais en même temps un autre qui sera 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 !

O.K., facile à dire, mais il faut savoir le faire derrière.

D'abord, créez le `TagType` pour notre entité `Tag`. Essayez de le faire vous-même, c'est vraiment simple, et une fois fini, vérifiez votre code avec le mien. Inspirez-vous de `ArticleType`, nul besoin de faire du par cœur.

Secret ([cliquez pour afficher](#))

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TagType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('nom'); // Notez ici que l'on n'a pas précisé le type de champ :
        // c'est parce que // le composant Form sait le reconnaître... depuis nos
        // annotations Doctrine !
    }

    public function getName()
    {
        return 'sdz_blogbundle_tagtype'; // N'oubliez pas de changer le nom du formulaire.
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class' => 'Sdz\BlogBundle\Entity\Tag', // Ni de modifier la classe ici.
        );
    }
}
```

Ensuite, il existe deux façons d'imbriquer des formulaires :

- avec une relation simple où un seul formulaire est imbriqué dans un autre. C'est le cas le plus courant, celui de notre `Client` avec une seule `Adresse` ;
- avec une relation multiple, où vous voulez imbriquer plusieurs fois le même formulaire dans un formulaire parent. C'est le cas de notre `Article`, par exemple, qui peut contenir plusieurs objets `Tag`.

Relation simple : imbriquer un seul formulaire

C'est le cas le plus courant, mais qui ne correspond malheureusement pas à notre exemple de l'article et ses tags. 🤔 Du coup, suivez bien, nous allons modifier **temporairement** notre entité `Article` pour remplacer tous les **tags** en **tag** : comme si l'on ne pouvait en fait attacher qu'un seul tag à un article. Voici ce que donne la modification :

Secret ([cliquez pour afficher](#))

Code : PHP - [Sélectionner](#)

```
<?php

namespace Sdz\BlogBundle\Entity;

use Sdz\BlogBundle\Entity\Tag;

class Article
{
    private $id;
    private $date;
    private $titre;
    private $contenu;
    private $pseudo;

    /**
     * Évidemment, nous devons modifier la définition de la relation : on passe à un ManyToOne !
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Tag")
     */
    private $tag;

    // ...

    public function setTag(Tag $tag) // Attention, ici, c'est setTag et non plus addTags.
    {
    }
```

```

    {
        $this->tag = $tag;
    }
    public function getTag()
    {
        return $this->tag;
    }
}

```



N'oubliez pas de mettre à jour la base de données avec la commande `php app/console doctrine:schema:update --force !`

Voilà, maintenant, nous pouvons imbriquer nos formulaires. C'est vraiment simple : allez dans `ArticleType` et rajoutez un champ « tag » (du nom de la propriété de notre entité), de type... `TagType`, bien sûr !

Code : PHP - [Sélectionner](#)

```

<?php
public function buildForm(FormBuilder $builder, array $options)
{
    $builder
        ->add('date', 'date')
        ->add('titre', 'text')
        ->add('contenu', 'textarea')
        ->add('pseudo', 'text')
        ->add('tag', new TagType)
    ;
}

```

Et voilà ! Allez sur la page d'ajout : [http://localhost/Symfony/web/app_dev.php\[...\]/outer/article](http://localhost/Symfony/web/app_dev.php[...]/outer/article). Le formulaire est déjà à jour, avec une partie « Tag » où l'on peut remplir le seul champ de ce formulaire, le champ nom. C'était d'une facilité déconcertante, n'est-ce pas ?

Essayez d'ajouter un `Article` avec son `Tag`. Alors ? Ça ne fonctionne pas, évidemment. Essayons de réfléchir un peu... Dans l'action du contrôleur, nous avons dit à l'*EntityManager* de persister l'article, mais on ne lui a rien précisé sur le tag contenu dans l'article (accessible via `<?php $article->getTag() ?>`, tout simplement). C'est pour cela que Symfony2 nous dit :

Citation : Symfony2

```

A new entity was found through the relationship 'Sdz\BlogBundle\Entity\Article#tag'
...
Explicitly persist the new entity

```

On va l'écouter et ajouter un `<?php persist() ?>` sur notre tag dans la méthode `<?php ajouterAction() ?>` de notre contrôleur :

Code : PHP - [Sélectionner](#)

```

<?php
// ...
public function ajouterAction()
{
    // ...
    // On l'enregistre dans la base de données.
    $em = $this->getDoctrine()->getEntityManager();
    $em->persist($article);
    $em->persist($article->getTag()); // Ici, on rajoute ce persist().
    $em->flush();
    // ...
}

```

Et voilà, cette fois-ci, le formulaire est pleinement opérationnel ; lancez-vous dans les tests !

Pour vérifier que cela fonctionne, rendez-vous sur la page de modification d'un article : [http://localhost/Symfony/web/app_dev.php\[...\]/de l'article](http://localhost/Symfony/web/app_dev.php[...]/de l'article).

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 va donc revenir à notre vrai cas où l'on peut ajouter plusieurs tags à un seul article. Tout d'abord, modifiez de nouveau l'entité `Article` pour remettre au pluriel les tags :

Secret ([cliquez pour afficher](#))

Code : PHP - [Sélectionner](#)

```

<?php

```



```

<?php

namespace Sdz\BlogBundle\Entity;

use Sdz\BlogBundle\Entity\Tag;

class Article
{
    private $id;
    private $date;
    private $titre;
    private $contenu;
    private $pseudo;

    /**
     * @ORM\ManyToMany(targetEntity="Sdz\BlogBundle\Entity\Tag")
     */
    private $tags;

    public function __construct()
    {
        $this->tags = new ArrayCollection(); // N'oubliez pas l'ArrayCollection.
        $this->date = new \Datetime();
    }

    // ...

    public function addTag(Tag $tag)
    {
        $this->tags[] = $tag;
    }
    public function getTags()
    {
        return $this->tags;
    }
}

```

Et bien sûr, mettez à jour la base de données.

Notre TagType ne va pas changer d'un poil. Un tag est un tag, qu'il soit ajouté une fois ou mille dans un article ne change rien pour lui. Voici un bel exemple du code découlé que nous permet de faire Symfony2 !

Par contre, notre ArticleType, lui, va changer. On va changer le champ « tag » de type TagType, en champ « tags » (toujours du nom de la propriété), de type « collection ». En fait, oui, ça pourrait être une liste (*collection*) de n'importe quoi. On va se servir des options du champ pour signaler que ce doit être une liste de plusieurs TagType. Voici ce que ça donne :

Code : PHP - [Sélectionner](#)

```

<?php
class ArticleType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
            ->add('date')
            ->add('titre')
            ->add('contenu')
        /**
         * Rappel :
         * - 1er argument : nom du champ ;
         * - 2e argument : type du champ ;
         * - 3e argument : tableau d'options du champ.
         */
            ->add('tags', 'collection', array('type' => new TagType))
        ;
    }
    // ...
}

```

Encore une fois, c'est vraiment tout ce qu'il y a à modifier côté formulaire. Vous pouvez d'ores et déjà observer le résultat. Pour cela, actualisez la page d'ajout d'un article. Ah mince, « Tags » 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, à partir du moment où vous pouvez ajouter plusieurs champs ou en supprimer, vous avez besoin de JavaScript pour le faire. Donc Symfony2 part du principe que de toute façon, vous gérerez ça avec du code JavaScript. O.K., ça ne nous fait pas peur !

D'abord, affichez la source de la page et regardez l'étrange balise `<div>` que Symfony2 a rajoutée :

Code : HTML - [Sélectionner](#)

```

<div id="sdz_blogbundle_articletype_tags" data-prototype="&lt;div&gt;
&lt;label class="required"&gt;$$$name$$$&lt;/label&gt;

&lt;div id="sdz_blogbundle_articletype_tags_$$$name$$$&gt;
&lt;div&gt;&lt;label for="sdz_blogbundle_articletype_tags_$$$name$$$_nom"
class="required"&gt;Nom&lt;/label&gt;
&lt;input type="text" id="sdz_blogbundle_articletype_tags_$$$name$$$_nom"
name="sdz_blogbundle_articletype[tags][$$$name$$$][nom]" required="required"

```

```
maxLength="255" value=""; /&gt;&lt;/div>&lt;/div>&lt;/div>">
</div>
```

Vous connaissez 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 notre champ « nom », 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/form.html.twig` :

Code : HTML - Sélectionner

```
<!-- Ajout d'un lien pour ajouter un champ tag supplémentaire. -->
<a href="#" id="add_tag">Ajouter un tag</a>

<!-- 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-1.6.2.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_tags');

    // On définit une fonction qui va ajouter un champ.
    function add_tag() {
        // 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(
            $($container.attr('data-prototype').replace(/\$\$name\$\$/g, index))
        );
    }

    // On ajoute un premier champ directement s'il n'en existe pas déjà un.
    if($container.children().length == 0) {
        add_tag();
    }

    // On ajoute un nouveau champ à chaque clic sur le lien d'ajout.
    $('#add_tag').click(function() {
        add_tag();
    });
});
</script>
```

Appuyez sur F5 sur la page d'ajout. Voilà qui est mieux !

Cependant, comme tout à l'heure, pour que le formulaire soit pleinement opérationnel, il nous faut adapter un peu le contrôleur pour qu'il persiste tous nos tags :

Code : PHP - Sélectionner

```
<?php
// ...
public function ajouterAction()
{
    // ...
    // On l'enregistre dans la base de données.
    $em = $this->getDoctrine()->getEntityManager();
    $em->persist($article);
    foreach($article->getTags() as $tag) // On persiste tous les tags de l'article.
    {
        $em->persist($tag);
    }
    $em->flush();
    // ...
}
```

Et voilà, votre formulaire est maintenant opérationnel ! Vous pouvez vous amuser à créer des articles contenant plein de tags en même temps. Vérifiez également depuis la page de modification, vous voyez que vous pouvez modifier les tags en direct. Vraiment pratique !



Pour information, ce champ de type *collection* que l'on vient de voir s'utilise avec n'importe quel autre type de champ, pas forcément un formulaire. Dans l'`ArticleType`, à la place du `<?php new TagType ?>`, vous auriez pu mettre « text » ou « file » ou n'importe quel autre type de champ classique.

Valider des objets

Never Trust User Input

Ce chapitre sur les formulaires va nous permettre d'introduire la validation des objets avec le service *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.

Si je mets ce paragraphe sur la validation dans le chapitre sur les formulaires, c'est que bien sûr, les deux sont 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 si un jour vous avez besoin de valider un de vos objets dans un contexte particulier, hors formulaire.

Le retour des annotations

La théorie de la validation

La théorie, très simple, est la suivante. On définit des règles de validation que l'on va rattacher à l'objet lui-même. Puis on fait appel à un service extérieur pour venir lire notre objet et ses règles, et définir si oui ou non l'objet en question respecte ces règles. Simple et logique.

Définir les règles de validation

Concrètement, et comme le titre l'indique, pour définir les règles de validation, on va utiliser les annotations. En fait, c'est justement pour cela que l'on avait choisi le format annotation pour définir les métadonnées de nos entités : nous allons avoir les métadonnées Doctrine et les règles de validation au même endroit ! Du coup, on centralise tout, et c'est très pratique à lire et à modifier.

Sans plus attendre, voici la syntaxe à respecter. Exemple avec notre objet `Article` :

Code : PHP - [Sélectionner](#)

```
<?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;

use Sdz\BlogBundle\Entity\Tag;

/**
 * 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()
```

```

... @Assert\NotBlank()
*/
    private $contenu;

    /**
     * @var string $pseudo
     *
     * @ORM\Column(name="pseudo", type="string", length=255)
     * @Assert\MinLength(5)
     */
    private $pseudo;

    /**
     * @ORM\ManyToOne(targetEntity="Sdz\BlogBundle\Entity\Tag")
     * @Assert\Valid()
     */
    private $tags;

    // ...
}

```

Vraiment pratique d'avoir les métadonnées Doctrine et les règles de validation au même endroit, n'est-ce pas ?

Bon, je ne reviens pas particulièrement sur ces règles, elles sont très intuitives. Le `<?php /* @Assert\Valid() */ ?>` est intéressant pour déclencher une sous-validation des objets Tag contenus. Il existe une flopée de règles de validation que je vous invite bien évidemment à aller consulter [dans la documentation](#). Je ne les traiterai pas ici, ce serait faire du copier-coller.

À titre d'entraînement, n'oubliez pas de définir les contraintes pour notre entité Tag.

Déclencher la validation

Comme je l'ai dit plus haut, ce n'est pas l'objet qui se valide tout seul. Ainsi, vous pouvez tout à fait faire un `<?php $article->setTitre('abc') ?>` (titre avec moins de 10 caractères) sans qu'aucune erreur ne se déclenche. Il faut utiliser le service *validator* pour cela. On lui demande de valider l'objet, et ce dernier nous retourne un tableau qui est soit vide si l'objet est valide, soit rempli des différentes erreurs lorsque l'objet n'est pas valide.

En pratique, on ne se servira que très peu du service *validator* nous-mêmes, car le formulaire de Symfony2 va le faire à notre place. Mais pour comprendre ce qu'il se passe dans les coulisses, voici comment on le ferait à la main :

Code : PHP - Sélectionner

```

<?php
// ...

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->setPseudo('Moi');         // Champ « pseudo » incorrect : moins de 3
    caractères.

    // On récupère le service validator.
    $validator = $this->get('validator');

    // On déclenche la validation.
    $errorList = $validator->validate($article);

    // Si le tableau n'est pas vide, on affiche les erreurs.
    if(count($errorList) > 0)
    {
        return new Response(print_r($errorList, true));
    }
    else
    {
        return new Response("L'article est valide !");
    }
}

```

Voilà pour la théorie. Encore une fois, vous ne vous servirez que rarement de cette syntaxe. Ce que l'on va utiliser le plus souvent, eh bien en fait, on l'a déjà utilisé ! C'est tout simplement lorsque vous appelez la méthode `<?php isValid() ?>` de votre formulaire !

Dans ce `<?php isValid() ?>`, le formulaire va se servir tout seul du service *validator* et valider comme il faut l'objet tel qu'hydraté par le visiteur. Bref, tout se fait dans les coulisses, mais maintenant, au moins, vous savez ce qu'il s'y fait. 😊

À retenir

- La validation est indépendante du formulaire.
- Les règles se définissent dans l'objet, en annotation ou tout autre format de votre choix.
- La liste des contraintes existantes est bien remplie et se trouve dans la documentation.
- Vous pouvez créer vos propres contraintes en suivant la documentation à ce sujet.

Application : les formulaires de notre blog

Nous n'avons qu'un seul formulaire pour l'instant, celui pour créer et modifier l'objet `Article`. Comme nous avons déjà notre `ArticleType`, vous n'avez plus qu'à vous occuper de la gestion du formulaire au niveau de notre `BlogController` dans les méthodes `<?php ajouterArticle() ?>` et `<?php modifierArticle() ?>`.



Au boulot ! Essayez d'implémenter vous-même la gestion du formulaire dans les actions correspondantes. Ensuite seulement, lisez la suite de ce paragraphe pour avoir la solution.

Entité `Article`, j'ai juste modifié le constructeur afin de pré-remplir la date directement lorsque l'on instancie l'objet :

Code : PHP - [Sélectionner](#)

```
<?php
public function __construct()
{
    $this->tags = new ArrayCollection();
    $this->date = new \Datetime();
}

// Ainsi, avec $article = new Article();, $article->date vaut déjà la date d'aujourd'hui.
```

Action « ajouter » :

Code : PHP - [Sélectionner](#)

```
<?php

use Sdz\BlogBundle\Entity\Article;
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' )
    {
        $form->bindRequest($request);
        if( $form->isValid() )
        {
            $em = $this->getDoctrine()->getEntityManager();

            $em->persist($article);
            foreach($article->getTags() as $tag)
            {
                $em->persist($tag);
            }
            $em->flush();

            return $this->redirect( $this->generateUrl('sdzblog') );
        }
    }

    return $this->render('SdzBlogBundle:Blog:ajouter.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

Action « modifier » :

Code : PHP - [Sélectionner](#)

```
<?php

use Sdz\BlogBundle\Entity\Article;
use Sdz\BlogBundle\Form\ArticleType;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
```

```
// ...

public function modifierAction($id)
{
    $em = $this->getDoctrine()->getEntityManager();

    // On vérifie que l'article d'id $id existe bien, sinon, erreur 404.
    if( ! $article = $em->getRepository('Sdz\BlogBundle\Entity\Article')->find($id) )
    {
        throw new NotFoundException('Article[id='.$id.'] inexistant');
    }

    $form = $this->createForm(new ArticleType(), $article);

    $request = $this->get('request');
    if( $request->getMethod() == 'POST' )
    {
        $form->bindRequest($request);
        if( $form->isValid() )
        {
            $em->persist($article);
            foreach($article->getTags() as $tag)
            {
                $em->persist($tag);
            }
            $em->flush();

            return $this->redirect( $this->generateUrl('sdzblog') );
        }
    }

    return $this->render('SdzBlogBundle:Blog:modifier.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

Voilà ! Ce chapitre important n'était pas si compliqué dans le fond !

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. 😊

Partie 3 : Les Cookbooks

Les *cookbooks* sont des petites et moyennes astuces qui vous permettent de réaliser des choses précises dans votre projet Symfony2. Comme ils couvrent souvent plusieurs notions en même temps, il est impossible de les intégrer dans le tutoriel, c'est pourquoi une partie leur est dédiée. Je vous invite donc à en lire un 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 ? 😊

La 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 - [Sélectionner](#)

```
<?php
public function viewAction($id)
{
    if( ! $article = $this->get('doctrine')->getEntityManager()-
>getRepository('Sdz\BlogBundle\Entity\Article')->find($id) )
    {
        throw new NotFoundException(sprintf('L\'article id:%s" n\'existe pas.', $id));
    }

    // votre vrai code
```

```

        return $this->render( 'SdzBlogBundle:Article:view.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'évènement **kernel.controller**. Cet évènement est déclenché lorsque le contrôleur sait quel contrôleur appeler (après le routeur, donc), mais avant d'exécuter effectivement le contrôleur. Ainsi, lors de cet évènement, 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.

La 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 - [Sélectionner](#)

```

MaRoute:
    pattern:  /view/{id}
    defaults: { _controller: SdzBlogBundle:Article:view }

```

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/ArticleController.php`).

- Avant :

Code : PHP - [Sélectionner](#)

```
<?php public function viewAction($id)
```

- Après :

Code : PHP - [Sélectionner](#)

```

<?php
// En haut du fichier

use Sdz\BlogBundle\Entity\Article;

// Et dans la classe
public function viewAction( Article $article )

```

Voici le code complet du contrôleur :

Code : PHP - [Sélectionner](#)

```
<?php
use Sdz\BlogBundle\Entity\Article;

// ...

public function viewAction( Article $article )
{
    // Récupérons le titre de l'article.
    $article->getTitre();

    return $this-
>render( 'SdzBlogBundle:Article:view.html.twig', array( 'article' => $article));
}
```

Et voilà ! Vous pouvez maintenant utiliser directement `<?php $article->getTitre() ?>` ou tout autre code utilisant `<?php $article ?>`. Magique, 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.

La 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 - [Sélectionner](#)

```
<!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 y 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.

La liste des *cookbooks* n'est pas définitive et ne le sera jamais : j'en rajouterai 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 !

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.

