symblog
creating a blog in Symfony2

[Partie 3] - Le modèle d'article : utilisation de Doctrine 2 et des données factices

> Je propose également des formations en petits groupes sur 2 à 3 jours, plus d'infos sur la page dédiée. N'hésitez pas à me contacter (06.62.28.01.87 ou clement [@] keiruaprod.fr) pour en discuter !

Introduction

Dans ce chapitre, nous allons commencer à explorer le modèle d'article. Ce modèle sera implémenté en utilisant l'ORM (pour Object Relation Mapper, soit `Lien Objet-Relation`) Doctrine 2 . Doctrine 2 nous permet de faire persister nos objets PHP. Il propose également un dialecte SQL personnel appelé DQL (pour Doctrine Query Language, ou `Langage de requêtes de Doctrine`). En plus de Doctrine 2, nous allons également aborder le concept de données factices (data fixtures). C'est un mécanisme permettant de peupler la base de donnée de l'environnement de développement et de test avec des données de test adéquates. A la fin de ce chapitre nous aurons défini le modèle d'article, mis à jour la base de donnée afin qu'elle reflète ce changement, et créé des articles factices. Nous aurons également construit les bases de la page d'affichage des article.

Doctrine 2: Le modèle

Afin que notre blog fonctionne, il nous faut un moyen de faire persister les données. Doctrine 2 fournit une librairie d'ORM conçue exactement dans cette optique. Doctrine 2 est conçu au dessus d'une couche d'abstraction de base de donnée très puissante qui la rend indépendante de la base de donnée utilisée : cela permet d'utiliser différents moteurs de stockage tels que MySQL, PostgreSQL ou SQLite. Nous allons utiliser MySQL dans ce tutorial, mais n'importe quel moteur peut être utilisé à la place.

> **Tip**
>
> Si vous n'êtes pas familier avec les ORM, nous allons en détailler le principe. La définition de Wikipedia dit:
>
> "Un mapping objet-relationnel (en anglais object-relational mapping ou ORM) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé. On pourrait le désigner par `` correspondance entre monde objet et monde relationnel ``"
>
> Ce qu'un ORM facilite, c'est la traduction des données d'une base de donnée relationnelle en des objets PHP que l'on peut manipuler. Cela permet d'encapsuler des opérations que l'on souhaite réaliser sur une table à l'intérieur d'une classe. Prenons l'exemple d'une table pour gérer les utilisateurs. Elle contiendra probablement des champs tels que le nom d'utilisateur, son mot de passe, son nom et sa date de naissance. L'orm va nous permettre d'appeller des méthodes `getUsername()` ou `setPassword()` dans la classe PHP. Les ORM vont bien plus loin que cela néanmoins, ils permettent de retrouver des entités liées pour nous, soit au moment où l'on charge l'entité utilisateur, soit de manière retardée par la suite. Maintenant imaginons que notre utilisateur a des amis qui lui sont liés. Il peut y avoir une table d'amis, où est stockée la clée primaire de la table utilisateur. L'ORM nous permet d'utiliser une méthode telle que `$user->getFriends()` pour récupérer les objets de la table d'amis. Si cela ne suffit pas, l'ORM se charge également de la persistance, ce qui nous permet de créer des objets PHP, d'appeler une méthode de sauvegarde (du genre `save()`), et de laisser l'ORM s'occuper des détails pour la sauvegarde dans la base de données. Comme nous allons utiliser Doctrine 2 comme librairie d'ORM, vous allez devenir plus à l'aise avec cette notion au cours de ce tutoriel.

> **Note**
>
> Bien que dans ce tutorial nous utilisions Doctrine 2 comme librairie d'ORM, vous pourriez opter pour la libraire Doctrine 2 ODM. il y a un certain nombre de variations de cette librairie, ce qui inclut des implémentations pour MongoDB et CouchDB. Regardez la page de projets Doctrine pour plus d'informations.
>
> Il y a également un article dans le cookbook qui explique comment mettre en place ODM avec Symfony2.

L'entité d'article

Nous allons commencer par créer la classe entité `Blog`. NDT: c'est le nom qu'a choisi l'auteur original de l'article pour la classe qui sert à modéliser un article. Nous avons déjà parlé des entités dans le chapitre précédent lorsque nous avons créé l'entité `Enquiry`. Comme le but d'une entité est de stocker

des données, il est logique d'en utiliser une pour représenter le contenu d'un article. En créant une entité, nous allons automatiquement lier ces données avec la base de données. Avec l'entité `Enquiry`, nous nous sommes contenté d'utiliser les informations par email au webmaster.

Créez un fichier dans `src/Blogger/BlogBundle/Entity/Blog.php` et collez-y le contenu suivant.

```
<?php
// src/Blogger/BlogBundle/Entity/Blog.php

namespace Blogger\BlogBundle\Entity;

class Blog
{
    protected $title;

    protected $author;

    protected $blog;

    protected $image;

    protected $tags;

    protected $comments;

    protected $created;

    protected $updated;
}
```

Comme vous pouvez le voir, il s'agit d'une simple classe PHP. Elle n'a ni classe parente, ni accesseurs. Les membres sont tous déclarés en `protected`, il est donc impossible d'accéder à eux lorsque l'on traite avec une instance de cette classe. Nous pourrions écrire nous même les accesseurs, mais Doctrine 2 propose une commande capable de s'en charger. En même temps, écrire des accesseurs n'est pas l'aspect le plus passionnant du projet.

Avant de lancer cette commande, il faut expliquer à Doctrine 2 comment l'entité `Blog` doit être associée à la base de donnée. Cela se fait via des métadonnées qui peuvent être définis dans plusieurs formats: `YAML`, `PHP`, `XML` et `Annotations`. Nous allons utiliser les annotations dans ce tutoriel. Il est important de noter que tous les membres de l'entité n'ont pas besoin d'être persistés, nous ne préciserons donc pas de métadonnées pour ceux qui sont dans cette situation, ce qui nous donne la flexibilité de choisir les informations à envoyer à la base de données. Remplacez le contenu de la classe `Blog` situé dans `src/Blogger/BlogBundle/Entity/Blog.php` par le suivant :

```
<?php
// src/Blogger/BlogBundle/Entity/Blog.php

namespace Blogger\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="blog")
 */
class Blog
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string")
     */
    protected $title;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $author;

    /**
     * @ORM\Column(type="text")
     */
    protected $blog;

    /**
     * @ORM\Column(type="string", length="20")
     */
    protected $image;

    /**
     * @ORM\Column(type="text")
     */
    protected $tags;
```

```
    protected $comments;

    /**
     * @ORM\Column(type="datetime")
     */
    protected $created;

    /**
     * @ORM\Column(type="datetime")
     */
    protected $updated;
}
```

Tout d'abord, on importe et crée un alias pour l'espace de nom de Doctrine 2. Cela nous permet d'utiliser les `annotations` pour décrire les métadonnées des entités. Les métadonnées nous fournissent des informations sur la manière dont les membres sont représentés dans la base de donnée.

> **Tip**
>
> Nous venons seulement de voir un petit sous ensemble des types d'association que propose Doctrine 2. Une liste complète est disponible sur le site web de Doctrine 2. Nous allons utiliser d'autres types d'association plus tard dans ce tutoriel.

L'oeil averti aura sûrement remarqué que l'attribut `$comments` n'a pas de métadonnées associées. Nous ne souhaitons pas le faire persister dans la base de données. En effet, il fournit seulement une liste des commentaires relatifs à un article. Si l'on pense en terme d'objet et non de base de données, cela prend tout son sens, comme vous pouvez le voir dans le bout de code suivant :

```
// Create a blog object.
$blog = new Blog();
$blog->setTitle("symblog - A Symfony2 Tutorial");
$blog->setAuthor("dsyph3r");
$blog->setBlog("symblog is a fully featured blogging website ...");

// Create a comment and add it to our blog
$comment = new Comment();
$comment->setComment("Symfony2 rocks!");
$blog->addComment($comment);
```

La portion de code ci-dessus illustre le comporte normal que l'on pourrait souhaiter d'une classe d'article et de commentaires. En interne, la méthode `$blog->addComment()` pourrait être implémentée comme ceci :

```
class Blog
{
    protected $comments = array();

    public function addComment(Comment $comment)
    {
        $this->comments[] = $comment;
    }
}
```

La méthode `addComment` se contente d'ajouter un objet `Comment` à la variable membre `$comments` de l'article. Récupérer les commentaires est alors très simple :

```
class Blog
{
    protected $comments = array();

    public function getComments()
    {
        return $this->comments;
    }
}
```

Comme on le voit, le membre `$comments` est simplement une liste d'objets `Comment`. Doctrine 2 ne change pas cette manière de fonctionner, mais va être capable de remplir automatiquement ce champ à partir de l'objet `blog`.

Maintenant que nous avons dit à Doctrine 2 comment associer les entités membres, voyons comment générer les accesseurs :

```
$ php app/console doctrine:generate:entities Blogger
```

Après avoir lancé la commande précédente, vous aurez remarqué que l'entité `Blog` a été mise à jour avec l'ajout des accesseurs. A chaque fois que nous allons faire des changements aux métadonnées de l'ORM, il va falloir relancer cette commande pour mettre à jour les accesseurs. Ceux qui existent déjà ne seront pas modifiés, donc les accesseurs existants ne seront pas remplacés par cette commande, c'est important si jamais vous souhaitez personnaliser par les suite les accesseurs.

> **Tip**

Bien que nous ayons utilisé les `annotations` dans notre entité, il est possible de convertir les informations de mapping dans un autre format en utilisant la commande `doctrine:mapping:convert`.

Par exemple, la commande suivante va convertir les associations dans les entités ci-dessus au format `yaml`.

```
$ php app/console doctrine:mapping:convert --namespace="Blogger\BlogBundle\Entity\Blog" yaml src/Blogger/BlogBundle/Resources/config/doctr
```

Cela va créer un fichier dans `src/Blogger/BlogBundle/Resources/config/doctrine/Blogger.BlogBundle.Entity.Blog.orm.yml` qui va contenir les mappings en `yaml` de l'entité `blog`.

La base de données

Création de la base de données

Si vous avez suivi le chapitre 1 de ce tutoriel, vous avez dû utiliser l'outil de configuration web pour rentrer les paramètres de la base de donnée. Si vous ne l'avez pas fait, mettez à jour les options `database_*` dans le fichier de configuration `app/parameters.ini`.

Il est maintenant temps de créer la base de donnée en utilisant une autre commande Doctrine 2. Cette commande crée seulement la base de données, mais pas les tables à l'intérieur. Si une base de donnée du même nom existe déjà, une erreur sera affichée et la base de donnée existante restera intacte.

```
$ php app/console doctrine:database:create
```

Nous sommes maintenant prêts pour créer la représentation de l'entité `Blog` dans la base de données. Il y a 2 moyens pour faire cela. Nous pouvons utiliser la commande `schema` de Doctrine 2 pour mettre à jour la base de donnée, ou bien les nettement plus puissantes migrations de Doctrine 2. Pour le moment, contentons nous de la commande `schema`, les migrations seront abordées dans un chapitre ultérieur.

Création de la table d'article

Pour créer la table `blog` dans notre base de données, on peut lancer la commande doctrine suivante :

```
$ php app/console doctrine:schema:create
```

Cela exécute le code SQL nécessaire à la génération du schéma de la base de donnée pour l'entité `blog`. Vous pouvez également ajouter l'argument `--dump-sql` optionnellement afin d'afficher le code SQL généré. Si vous regardez le contenu de votre base de données, vous pourrez voir que la table `blog` a été créée, avec des champs qui correspondent à ce que nous avions spécifié.

Tip

Nous avons utilisé un certain nombre de lignes de commandes Symfony2 jusqu'à présent, et dans une vrai console le format de commande permet toujours d'obtenir de l'aide en ajoutant l'option `--help`. Symfony2 n'échappe pas à cette règle: pour voir l'aide relative à la commande `doctrine:schema:create`, exécutez la ligne suivante :

```
$ php app/console doctrine:schema:create --help
```

Les informations d'aide vont alors afficher l'usage et les options disponible. La plupart des commandes proposent un grand nombre d'options permettant de personnaliser l'exécution d'une commande.

Intégration du Modèle avec la Vue : affichage d'un article

Maintenant que l'entité `Blog` a été créée et que la base de donnée le reflète, nous pouvons commencer à intégrer le modèle dans la vue. Nous allons commencer par construire la page d'affichage des articles de notre blog.

La route d'affichage d'un article

Nous allons commencer par créer une route pour l'action `show`. Un article va être caractérisé par un identifiant unique, cet identifiant se doit donc d'être présent dans l'URL. Mettez à jour les règles de routage du `BloggerBlogBundle` dans `src/Blogger/BlogBundle/Resources/config/routing.yml` en y ajoutant ce qui suit: with the following

```
# src/Blogger/BlogBundle/Resources/config/routing.yml
BloggerBlogBundle_blog_show:
    pattern:  /{id}
    defaults: { _controller: BloggerBlogBundle:Blog:show }
    requirements:
        _method:  GET
        id: \d+
```

Comme l'identifiant de l'article sera présent dans l'URL, nous avons ajouté un élément `id` dans la route. Sans plus de détail, cela signifie que les adresses `http://symblog.co.uk/1` et `http://symblog.co.uk/my-blog` valident toutes les deux la route. Comme nous savons que l'identifiant est un entier (c'est ce que l'on a défini dans le mapping), on peut ajouter une contrainte qui ne valide la route que si le paramètre `id` est un entier. C'est réalisé grâce à la ligne `id: \d+` dans la section `requirements`, qui définit les conditions à valider. Maintenant, seule la première adresse serait valide. Vous pouvez également voir que lorsque l'adresse valide cette route, c'est la méthode `show` du contrôleur `Blog` du `BloggerBlogBundle` qui est executée. Il ne reste plus qu'à créer le contrôleur `Blog`, c'est ce que nous allons faire tout de suite.

L'action `Show` du Contrôleur

Le lien entre le Modèle et la Vue, c'est le Contrôleur, c'est donc là que nous allons commencer à créer la page d'affichage. Nous pourrions ajouter l'action `show` à notre contrôleur `Page` déjà existant, mais comme cette page se contente d'afficher les entités `blog`, cela a plus de sens de le mettre dans un contrôleur à part.

Créez un nouveau fichier dans `src/Blogger/BlogBundle/Controller/BlogController.php` et collez-y le code suivant :

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

namespace Blogger\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

/**
 * Blog controller.
 */
class BlogController extends Controller
{
    /**
     * Show a blog entry
     */
    public function showAction($id)
    {
        $em = $this->getDoctrine()->getEntityManager();

        $blog = $em->getRepository('BloggerBlogBundle:Blog')->find($id);

        if (!$blog) {
            throw $this->createNotFoundException('Unable to find Blog post.');
        }

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

Nous avons créé un nouveau contrôleur pour l'entité `Blog`, et y avons défini une action `show`. Comme nous avons spécifié un paramètre `id` pour la route `BloggerBlogBundle_blog_show`, ce paramètre sera passé en argument à la méthode `showAction`. Si nous avions passé plusieurs paramètres dans la règle de routage, ils auraient également été fournis sous la forme d'arguments séparés.

> **Tip**
>
> Les actions du contrôleur fournissent également un objet de type `Symfony\Component\HttpFoundation\Request` si vous le spécifiez parmi les paramètres. Cela peut être utile lorsque l'on traite avec les formulaires. Nous en avons déjà utilisé dans le chapitre 2, mais nous ne nous sommes pas servis de cette méthode car nous avons utilisé une des méthodes d'aide du contrôleur de base `Symfony\Bundle\FrameworkBundle\Controller\Controller` comme suit :
>
> ```php
> // src/Blogger/BlogBundle/Controller/PageController.php
> public function contactAction()
> {
>     // ..
>     $request = $this->getRequest();
> }
> ```
>
> Nous aurions très bien pu écrire ce code de la manière suivante :
>
> ```php
> // src/Blogger/BlogBundle/Controller/PageController.php
>
> use Symfony\Component\HttpFoundation\Request;
>
> public function contactAction(Request $request)
> {
>     // ..
> }
> ```
>
> Les deux méthodes réalisent la même tâche, mais si votre controlleur n'étendait pas la classe de base `Symfony\Bundle\FrameworkBundle\Controller\Controller`, vous ne pourriez pas utiliser la première méthode.

Il nous faut ensuite récupérer les entités `Blog` dans la base de données. Nous utilisons pour cela une seconde méthode de la classe `Symfony\Bundle\FrameworkBundle\Controller\Controller` pour obtenir le gestionnaire d'entités de Doctrine 2. Le but du gestionnaire d'entités est de récupérer les objets venant de la base de données, et de les y faire persister. Nous utilisons ensuite l'objet `EntityManger` pour obtenir le `Repository` de Doctrine2 pour l'entité `BloggerBlogBundle:Blog`. La syntaxe spécifiée ici est simplement un raccourci qui peut être utilisé avec Doctrine 2 au lieu de préciser le nom entier, c'est à dire `Blogger\BlogBundle\Entity\Blog`. Avec le dépot d'objets (le repository), nous appelons la méthode `find()` avec pour argument la variable `$id`. Cette méthode se charge de retrouver tous les objets à partir de leur clé primaire.

Enfin, nous vérifions qu'une entité a été trouvé, et fournissons cette entité à la vue. Si aucune entité n'est trouvée, une exception est lancée, qui va se charger de générer une `erreur 404`.

> **Tip**
>
> L'objet repository (le dépot d'objet) nous donne accès à un certain nombre de méthodes auxilliaires utiles, telles que :
>
> ```
> // Renvoit les entité dont l'attribut 'author' vaut 'dsyph3r'
> $em->getRepository('BloggerBlogBundle:Blog')->findBy(array('author' => 'dsyph3r'));
>
> // Renvoit une entité dont l'attribut 'slug' vaut 'symblog-tutorial'
> $em->getRepository('BloggerBlogBundle:Blog')->findOneBySlug('symblog-tutorial');
> ```
>
> Nous allons par la suite créer nos propres classes de repository dans le chapitre suivant, lorsque nous aurons besoin d'effectuer des requêtes plus complexes.

La vue

Maintenant que nous avons construit l'action `show` pour le contrôleur `Blog`, nous pouvons nous concenter sur l'affichage des entités `Blog`. Comme précisé dans l'action `show`, le template `BloggerBlogBundle:Blog:show.html.twig` sera affiché. Commençons par créer ce fichier, dans `src/Blogger/BlogBundle/Resouces/views/Blog/show.html.twig`, et ajoutons y le code qui suit :

```
{# src/Blogger/BlogBundle/Resouces/views/Blog/show.html.twig #}
{% extends 'BloggerBlogBundle::layout.html.twig' %}

{% block title %}{{ blog.title }}{% endblock %}

{% block body %}
    <article class="blog">
        <header>
            <div class="date"><time datetime="{{ blog.created|date('c') }}">{{ blog.created|date('l, F j, Y') }}</time></div>
            <h2>{{ blog.title }}</h2>
        </header>
        <img src="{{ asset(['images/', blog.image]|join) }}" alt="{{ blog.title }} image not found" class="large" />
        <div>
            <p>{{ blog.blog }}</p>
        </div>
    </article>
{% endblock %}
```

Comme vous l'attendiez, nous commençons par étendre le template principale du `BloggerBlogBundle`. Ensuite, on remplace le titre de la page pour avoir à la place celui de l'article. C'est utile pour le SEO (Search Engine Optimization: l'ensemble de techniques qui ont pour but d'améliorer les résultats dans les moteurs de recherche), car la page de titre de l'article décrit plus spécifiquement le contenu de cette page que le titre générique que nous avons mis par défaut. Enfin, on remplace le corps de la page pour afficher le contenu de l'entité `Blog`. Nous utilisons la fonction `asset` à nouveau pour afficher l'image de l'article. Les images devraient être placées dans le répertoire `web/images`.

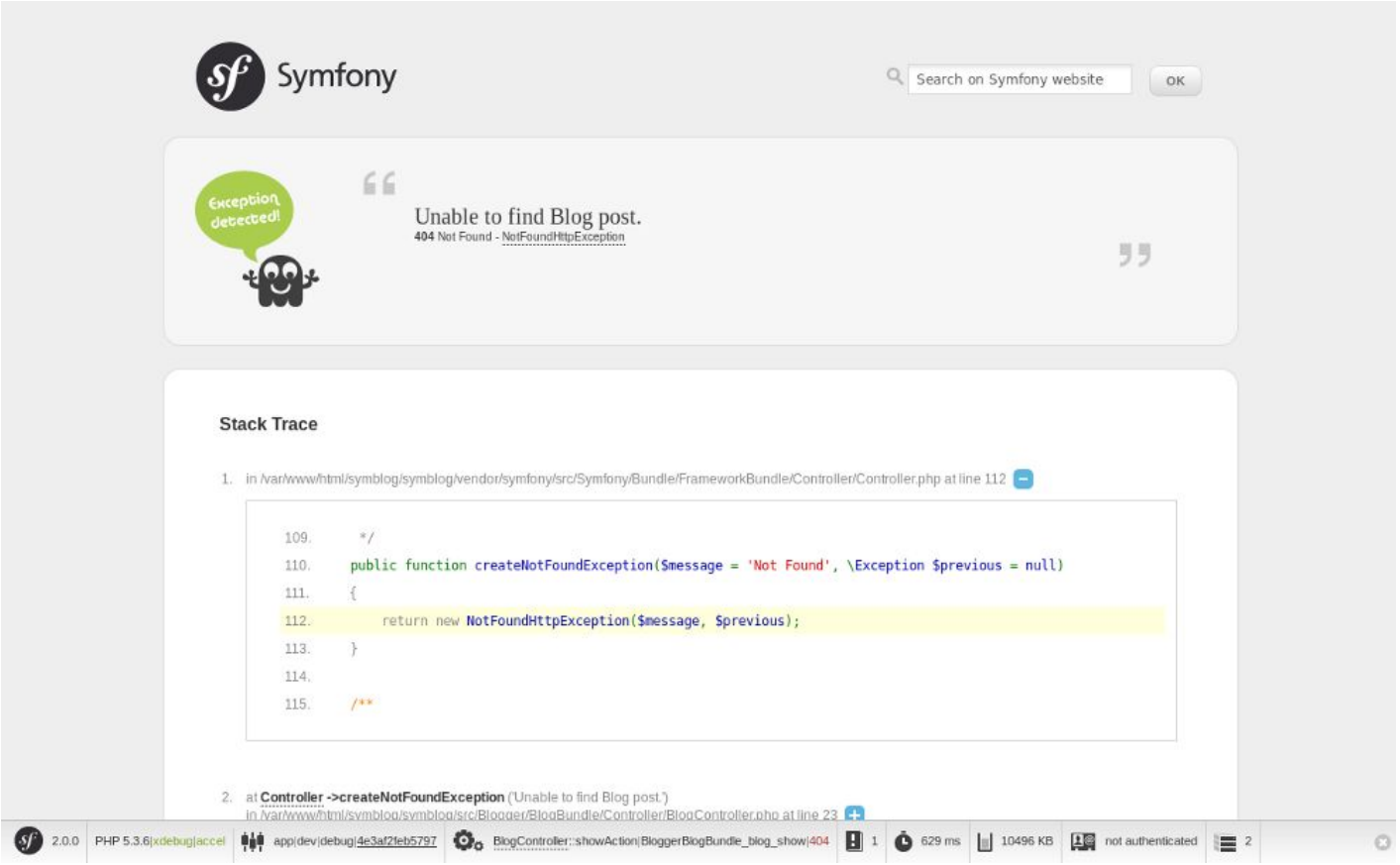
CSS

Afin que la page d'affichage des articles soit visuellement agréable, il faut lui ajouter du style. Mettez à jour la feuille de style dans `src/Blogger/BlogBundle/Resouces/public/css/blog.css` avec le contenu suivant :

```
.date { margin-bottom: 20px; border-bottom: 1px solid #ccc; font-size: 24px; color: #666; line-height: 30px }
.blog { margin-bottom: 20px; }
.blog img { width: 190px; float: left; padding: 5px; border: 1px solid #ccc; margin: 0 10px 10px 0; }
.blog .meta { clear: left; margin-bottom: 20px; }
.blog .snippet p.continue { margin-bottom: 0; text-align: right; }
.blog .meta { font-style: italic; font-size: 12px; color: #666; }
.blog .meta p { margin-bottom: 5px; line-height: 1.2em; }
.blog img.large { width: 300px; min-height: 165px; }
```

> **Note**
>
> Si vous n'utilisez pas la méthode symlink pour référencer les ressources utilisées dans le dossier `web`, vous devez la commande d'installation des ressource pour mettre à jour les changements qui ont eu lieu dans la feuille de style.
>
> ```
> $ php app/console assets:install web
> ```

Comme nous avons maintenant construit le contrôleur et la vue pour l'action `show`, allons jeter un oeil à la page que nous venons de créer. Rendez vous avec votre navigateur à l'adresse `http://symblog.dev/app_dev.php/1`. Ce n'est probablement pas la page que vous attendiez...



Symfony2 a généré une erreur 404. Comme il n'y a rien dans la base de données, il n'y a pas d'entité ayant pour `id` la valeur 1.

Vous pourriez simplement ajouter un élément dans la table `blog` de votre base de données, mais nous allons faire mieux. Nous servir de données factices, également appelées les `data fixtures`.

Données factices

On peut utiliser des fixtures pour remplir la base de donnée avec des données de test. Pour cela, nous allons utiliser l'extension `Doctrine Fixtures extension and bundle`. Cette extension n'est pas disponible de base avec l'édition standard de Symfony2, nous allons devoir l'installer manuellement. Heureusement, c'est facile à faire. Ouvrez le fichier `deps` à la racine du projet, et ajoutez y la nouvelle extension à la suite de celles déjà présentes et ajoutant ceci:

```
[doctrine-fixtures]
    git=http://github.com/doctrine/data-fixtures.git

[DoctrineFixturesBundle]
    git=http://github.com/symfony/DoctrineFixturesBundle.git
    target=/bundles/Symfony/Bundle/DoctrineFixturesBundle
```

Maintenant, mettez à jour les vendors pour que les changements soient pris en compte.

```
$ php bin/vendors install
```

Cela va télécharger les dernières versions disponible sur Github de chacun des bundles, et les installer au bon endroit.

> **Note**
>
> Si vous êtes sur une machine où Git n'est pas installé, vous devrez télécharger et installer manuellement l'extension.
>
> Pour l'extension doctrine-fixtures: Téléchargez la version actuelle disponible sur Github, et décompressez son contenu dans `vendor/doctrine-fixtures`.
>
> Pour le DoctrineFixturesBundle: Téléchargez la version actuelle disponible sur Github, et décompressez son contenu dans `vendor/bundles/Symfony/Bundle/DoctrineFixturesBundle`.

Mettez ensuite à jour le fichier `app/autoloader.php` pour enregistrer les nouveaux espaces de noms. Comme les `DataFixtures` sont également dans l'espace de nom `Doctrine\Common`, ils doivent être placé avant la directive `Doctrine\Common` déjà existante, puisqu'elle précisent un nouveau chemin. Les espaces de

noms sont vérifiés de haut en bas, donc les espaces de noms les plus précis doivent être enregistrés avant ceux qui le sont moins.

```
// app/autoloader.php
// ...
$loader->registerNamespaces(array(
// ...
'Doctrine\\Common\\DataFixtures'    => __DIR__.'/../vendor/doctrine-fixtures/lib',
'Doctrine\\Common'                  => __DIR__.'/../vendor/doctrine-common/lib',
// ...
));
```

Maintenant enregistrons le `DoctrineFixturesBundle` dans le noyeau situé dans `app/AppKernel.php`

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\DoctrineFixturesBundle\DoctrineFixturesBundle(),
        // ...
    );
    // ...
}
```

Articles factices

Nous sommes maintenant prêts à définir du contenu factice pour nos articles. Créez un fichier de `fixtures` dans `src/Blogger/BlogBundle/DataFixtures/ORM/BlogFixtures.php` et ajoutez-y le contenu suivant :

```
<?php
// src/Blogger/BlogBundle/DataFixtures/ORM/BlogFixtures.php

namespace Blogger\BlogBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Blogger\BlogBundle\Entity\Blog;

class BlogFixtures implements FixtureInterface
{
    public function load($manager)
    {
        $blog1 = new Blog();
        $blog1->setTitle('A day with Symfony2');
        $blog1->setBlog('Lorem ipsum dolor sit amet, consectetur adipiscing eletra electrify denim vel ports.\nLorem ipsum dolor sit amet, consectetur
        $blog1->setImage('beach.jpg');
        $blog1->setAuthor('dsyph3r');
        $blog1->setTags('symfony2, php, paradise, symblog');
        $blog1->setCreated(new \DateTime());
        $blog1->setUpdated($blog1->getCreated());
        $manager->persist($blog1);

        $blog2 = new Blog();
        $blog2->setTitle('The pool on the roof must have a leak');
        $blog2->setBlog('Vestibulum vulputate mauris eget erat congue dapibus imperdiet justo scelerisque. Na. Cras elementum molestie vestibulum. Morb
        $blog2->setImage('pool_leak.jpg');
        $blog2->setAuthor('Zero Cool');
        $blog2->setTags('pool, leaky, hacked, movie, hacking, symblog');
        $blog2->setCreated(new \DateTime("2011-07-23 06:12:33"));
        $blog2->setUpdated($blog2->getCreated());
        $manager->persist($blog2);

        $blog3 = new Blog();
        $blog3->setTitle('Misdirection. What the eyes see and the ears hear, the mind believes');
        $blog3->setBlog('Lorem ipsumvehicula nunc non leo hendrerit commodo. Vestibulum vulputate mauris eget erat congue dapibus imperdiet justo scele
        $blog3->setImage('misdirection.jpg');
        $blog3->setAuthor('Gabriel');
        $blog3->setTags('misdirection, magic, movie, hacking, symblog');
        $blog3->setCreated(new \DateTime("2011-07-16 16:14:06"));
        $blog3->setUpdated($blog3->getCreated());
        $manager->persist($blog3);

        $blog4 = new Blog();
        $blog4->setTitle('The grid - A digital frontier');
        $blog4->setBlog('Lorem commodo. Vestibulum vulputate mauris eget erat congue dapibus imperdiet justo scelerisque. Nulla consectetur tempus nisl
        $blog4->setImage('the_grid.jpg');
        $blog4->setAuthor('Kevin Flynn');
        $blog4->setTags('grid, daftpunk, movie, symblog');
        $blog4->setCreated(new \DateTime("2011-06-02 18:54:12"));
        $blog4->setUpdated($blog4->getCreated());
        $manager->persist($blog4);

        $blog5 = new Blog();
```

```
        $blog5->setTitle('You\'re either a one or a zero. Alive or dead');
        $blog5->setBlog('Lorem ipsum dolor sit amet, consectetur adipiscing elittibulum vulputate mauris eget erat congue dapibus imperdiet justo scele
        $blog5->setImage('one_or_zero.jpg');
        $blog5->setAuthor('Gary Winston');
        $blog5->setTags('binary, one, zero, alive, dead, !trusting, movie, symblog');
        $blog5->setCreated(new \DateTime("2011-04-25 15:34:18"));
        $blog5->setUpdated($blog5->getCreated());
        $manager->persist($blog5);

        $manager->flush();
    }

}
```

Ce fichier contient un certain nombre de choses importantes à savoir lorsque l'on utilise Doctrine 2, en particulier sur comment faire persister les entités dans la base de données.

Regardons comment on crée un article :

```
$blog1 = new Blog();
$blog1->setTitle('A day in paradise - A day with Symfony2');
$blog1->setBlog('Lorem ipsum dolor sit d us imperdiet justo scelerisque. Nulla consectetur...');
$blog1->setImage('beach.jpg');
$blog1->setAuthor('dsyph3r');
$blog1->setTags('symfony2, php, paradise, symblog');
$blog1->setCreated(new \DateTime());
$blog1->setUpdated($this->getCreated());
$manager->persist($blog1);
// ..

$manager->flush();
```

On commence par créer une instance de la classe `Blog`, et spécifie les valeurs pour ses attributs. A cet instant, Doctrine 2 ne connait rien de l'objet `Entity`. C'est seulement lors de l'appel de `$manager->persist($blog1)` que Doctrine 2 prend en charge les objets entité. L'objet `$manager` est ici une instance de `EntityManager` que nous avons vu plus tôt, lorsque nous allons chercher des objets entité dans la base de données. Il est important de noter que bien que Doctrine 2 soit désormais en charge de l'objet entité, cet objet n'est toujours pas persisté dans la base de donnée. Un appel à la méthode `$manager->flush()` est nécessaire pour cela. La méthode flush oblige Doctrine 2 à intéragir avec la base de donnée pour toute les entités dont il s'occupe. Par souci de performance, il est nécessaire de regrouper les appels Doctrine 2 et réaliser un unique flush, c'est comme ça que nous avons fait avec nos données factices. On crée chaque entité, on dit à Doctrine 2 qu'il en a la charge, et finalement on sauvegarde toutes les entités en une fois à la fin via `flush`.

Charger les données factices

Nous sommes maintenant prêt pour charger les données factices dans la base de données.

```
$ php app/console doctrine:fixtures:load
```

Si vous regardez la page `http://symblog.dev/app_dev.php/1`, vous devriez maintenant y voir un article.

Essayez de changer la valeur du paramètre `id` dans l'URL pour la valeur 2. Vous devriez alors voir l'article suivant.

Si toutefois vous vous rendez à l'adresse `http://symblog.dev/app_dev.php/100` vous devriez avoir une erreur 404, car il n'existe pas d'entité ayant pour id la valeur 100. Maintenant, essayez l'URL `http://symblog.dev/app_dev.php/symfony2-blog`. Pourquoi n'avons nous pas droit à une erreur 404 ? Car l'action show n'est jamais executée. L'url n'arrive pas à faire correspondre cette adresse à une règle de routage (à cause de la nécessité pour l'identifiant des articles d'être un entier), c'est pourquoi on a à la place une exception qui dit: il n'existe pas de route pour cette adresse - `No route found for "GET /symfony2-blog"`.

Les Timestamps

Le terme le plus proche de Timestamp en français étant l'infame `Horodatage`, je vais continuer d'utiliser le terme Timestamp. En gros, un timestamp, c'est un attribut qui sert à stocker une information sur une date. Pour finir ce chapitre, nous allons regarder les 2 timestamps de l'entité `Blog`; `created` et `updated`. Les fonctionnalités de ces 2 attributs sont communément définies comme `Timestampable`. Promis, je vais quand même continuer de faire un effort pour les traductions, mais ne m'obligez pas à dire `horodatables`. Ces attributs stockent les informations sur la date et l'heure auxquelles un article a été créé, puis mis à jour pour la dernière fois. Comme nous ne souhaitons pas mettre à jour ce champ manuellement à chaque création ou mise à jour d'article, nous allons nous reposer sur Doctrine.

Doctrine 2 propose un système d'évènements qui fournit des callback de cycle de vie.

On peut utiliser ces callback pour préciser que nos entités doivent être averties de certains évènements. Il est par exemple possible d'être prévenu avant la mise à jour d'une entité, après une sauvegarde ou avant la suppression d'une entité. Afin d'utiliser ces callback, il est nécessaire de marquer les entités, ce que l'on fait dans les métadonnées. Mettez à jour l'entité `Blog` dans `src/Blogger/BlogBundle/Entity/Blog.php` avec le contenu suivant :

```
<?php
// src/Blogger/BlogBundle/Entity/Blog.php

// ..

/**
 * @ORM\Entity
 * @ORM\Table(name="blog")
 * @ORM\HasLifecycleCallbacks()
 */
class Blog
{
    // ..
}
```

Maintenant ajoutons une méthode dans l'entité `Blog` qui enregistre l'événement `preUpdate`. Nous ajoutons également un constructeur pour définir les valeurs par défaut des attributs `created` et `updated`.

```
<?php
// src/Blogger/BlogBundle/Entity/Blog.php
```

```php
// ..

/**
 * @ORM\Entity
 * @ORM\Table(name="blog")
 * @ORM\HasLifecycleCallbacks()
 */
class Blog
{
    // ..

    public function __construct()
    {
        $this->setCreated(new \DateTime());
        $this->setUpdated(new \DateTime());
    }

    /**
     * @ORM\preUpdate
     */
    public function setUpdatedValue()
    {
       $this->setUpdated(new \DateTime());
    }

    // ..
}
```

On enregistre l'entité `Blog` afin d'être notifié de l'évènement `preUpdate`, utilisé pour mettre à jour la valeur de `updated`. Maintenant, en relançant la commande de chargement des données factices, vous allez voir que les valeurs des 2 attributs ont été affectées automatiquement.

> **Tip**
>
> Comme les attributs timestampables sont un besoin récurrent dans les entités, un bundle est apparu pour ajouter son support. Il s'agit du StofDoctrineExtensionsBundle, qui fournit plusieurs extensions pour Doctrine 2 intéressante comme Timestampable, Sluggable, and Sortable (triable).
>
> Nous verrons comment intégrer ce bundle plus loin dans le tutoriel. Les plus pressés peuvent déjà regarder la page du cookbook à ce sujet.

Conclusion

Nous avons couvert un certain nombre de concepts qui traitent du Modèle avec Doctrine 2. Nous avons également regardé comment générer des données factices, qui propose une solution simple pour avoir des données de test pour la période de développement et de test.

La prochaine fois, nous regarderons comment étendre le modèle pour y ajouter le support des commentaires. Nous allons également commencer à construire la page d'accueil, et construire un dépôt personnalisé pour cela. Nous parlerons également des migrations avec Doctrine, ainsi que des intéractions entre les formulaires et cette librairie pour permettre l'ajout de commentaires aux articles.