symblog

creating a blog in Symfony2

[Partie 4] - Le modèle de commentaires : ajouter des commentaires, dépôts Doctrine 2 et migrations.

> 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 améliorer les articles que nous avons créés au chapitre précédent en ajoutant la possibilité d'y mettre des commentaires. Nous allons pour cela créer le modèle de commentaires, qui va stocker les commentaires de chaque article. Nous allons également parler des relations entre les modèles, car un article peut en effet contenir plusieurs commentaires. Nous utiliserons les dépôts ainsi que le moteur de construction de requêtes de Doctrine 2 pour récupérer les entités depuis la base de données. Nous allons également évoquer le thème des migrations Doctrine 2, qui permettent, par la programmation, de déployer des changements dans une base de donnée. A la fin de ce chapitre, nous aurons créé le modèle de commentaires, que nous aurons lié à celui des articles. Nous aurons également mis à jour la page d'accueil, et aurons fourni aux utilisateurs la possibilité de commenter les articles.

La page d'accueil

Commençons par construire la page d'accueil. Comme tout blog qui se respecte, il faut afficher un bout de chaque article, du plus récent au plus ancien. L'article complet sera disponible par un lien vers une page à cet effet. Comme nous avons déjà construit une route pour l'affichage d'un article, et que nous disposons d'un contrôleur et d'une vue pour la page d'accueil, il suffit de les mettre à jour.

Récupérer un article : requête sur le modèle

Afin d'afficher les articles, nous devons les récupérer depuis la base de donnée. Doctrine 2 utilise le langage langage de requêtes Doctrine (pour Doctrine Query Language, ou DQL) ainsi qu'un système de construction de requêtes (QueryBuilder) pour celà. Vous pouvez bien évidemment utiliser du SQL pûr avec Doctrine 2, mais c'est fortement découragé, car celà retire l'abstraction que Doctrine nous fournit. Nous allons utiliser le `QueryBuilder`, car il nous fournit une manière objet sympathique pour effectuer nos requêtes sur la base de donnée. Nous allons mettre à jour l'action `index` du contrôleur `Page` dans `src/Blogger/BlogBundle/Controller/PageController.php` pour récupérer les articles de la base de donnée.

```
// src/Blogger/BlogBundle/Controller/PageController.php
class PageController extends Controller
{
    public function indexAction()
    {
        $em = $this->getDoctrine()
                   ->getEntityManager();

        $blogs = $em->createQueryBuilder()
                    ->select('b')
                    ->from('BloggerBlogBundle:Blog',  'b')
                    ->addOrderBy('b.created', 'DESC')
                    ->getQuery()
                    ->getResult();

        return $this->render('BloggerBlogBundle:Page:index.html.twig', array(
            'blogs' => $blogs
        ));
    }

    // ..
}
```

On commence par obtenir une instance du `QueryBuilder` à partir de `EntityManager`. Celà nous permet de commencer à construire la requête à partir des nombreuses méthodes que le `QueryBuilder` propose. Une liste complète de ces méthodes est disponible dans la documentation du `QueryBuilder`. Un bon point de départ, c'est regarder les méthodes d'assistance méthodes d'assistance. Il s'agit des méthodes que nous allons utiliser, tel que `select()`, `from()` et `addOrderBy()`. Comme avec les intéractions précédentes avec Doctrine 2, nous pouvons utiliser la notation raccourcie pour faire référence à l'entité `Blog` via `BloggerBlogBundle:Blog` (souvenez vous que c'est la même chose que mettre `Blogger\BlogBundle\Entity\Blog`). Une fois qu'on a fini de spécifier les critères de la requête, on appelle `getQuery()` qui renvoit une instance de `DQL`. Nous ne pouvons pas obtenir de résultats depuis l'objet `QueryBuilder`: il faut passer par une instance de `DQL` d'abord, qui propose une méthode `getResult()` en charge de nous renvoyer une liste d'entités de `Blog`. Nous verrons par la suite que cette instance `DQL` propose plusieurs méthodes pour renvoyer les résultats tels que `getSingleResult()` et `getArrayResult()`.

La vue

Maintenant que nous avons une liste d'entité `Blog`, il faut les afficher. Remplacez le contenu du template de la page d'accueil situé dans `src/Blogger/BlogBundle/Resources/views/Page/index.html.twig` par ce qui suit :

```
{# src/Blogger/BlogBundle/Resources/views/Page/index.html.twig #}
{% extends 'BloggerBlogBundle::layout.html.twig' %}

{% block body %}
    {% for blog in blogs %}
        <article class="blog">
            <div class="date"><time datetime="{{ blog.created|date('c') }}">{{ blog.created|date('l, F j, Y') }}</time></div>
            <header>
                <h2><a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id }) }}">{{ blog.title }}</a></h2>
            </header>

            <img src="{{ asset(['images/', blog.image]|join) }}" />
            <div class="snippet">
                <p>{{ blog.blog(500) }}</p>
                <p class="continue"><a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id }) }}">Continue reading...</a></p>
            </div>

            <footer class="meta">
                <p>Comments: -</p>
                <p>Posted by <span class="highlight">{{blog.author}}</span> at {{ blog.created|date('h:iA') }}</p>
                <p>Tags: <span class="highlight">{{ blog.tags }}</span></p>
            </footer>
        </article>
    {% else %}
        <p>There are no blog entries for symblog</p>
    {% endfor %}
{% endblock %}
```

Nous utilisons ici une des structures de contrôle de Twig, la structure `for..else..endfor`. Si vous n'avez pas encore utilisé de moteur de template, vous reconnaitrez peut être ce genre de bout de code :

```
<?php if (count($blogs)): ?>
    <?php foreach ($blogs as $blog): ?>
        <h1><?php echo $blog->getTitle() ?><?h1>
        <!-- rest of content -->
    <?php endforeach ?>
<?php else: ?>
    <p>There are no blog entries</p>
<?php endif ?>
```

La structure de contrôle `for..else..endfor` de Twig est une manière bien plus propre de réaliser ceci. La plupart du code dans le template de la page d'accueil se charge d'afficher les informations sur l'article en HTML. Néanmoins, il y a plusieurs points à noter. Tout d'abord, nous utilisons la fonction Twig `path` pour générer l'adresse vers la page d'affichage des articles. Comme la route a besoin d'un `id` dans l'URL pour être générée, nous le passons en argument, comme dans l'exemple suivant :

```
<h2><a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id }) }}">{{ blog.title }}</a></h2>
```

Ensuite, nous affichons le contenu de l'article avec `<p>{{ blog.blog(500) }}</p>`. La valeur `500` que nous fournissons en argument est la longueur maximum de l'article que nous voulons afficher. Afin que cela fonctionne, nous devons mettre à jour la méthode `getBlog` que Doctrine 2 a généré pour nous. Mettez à jour la méthode `getBlog` de l'entité `Blog` dans `src/Blogger/BlogBundle/Entity/Blog.php` avec ce qui suit :

```
// src/Blogger/BlogBundle/Entity/Blog.php
public function getBlog($length = null)
{
    if (false === is_null($length) && $length > 0)
        return substr($this->blog, 0, $length);
```

```
    else
        return $this->blog;
}
```

Comme le comportement habituel de la méthode `getBlog` est de renvoyer le contenu complet de l'article, on définit une valeur par défaut pour le paramètre `$length` à `null`. Si le paramètre `null` est passé en paramètres, le contenu complet de l'article est affiché.

Si vous vous rendez maintenant à l'adresse `http://symblog.dev/app_dev.php/`, vous devriez voir que la page d'accueil affiche les derniers articles du blog. Vous devriez également pouvoir naviguer vers les articles complets en cliquant sur leur titre ou sur le lien 'continue reading...'.

Page d'accueil de symblog

Bien que nous pouvons effectuer nos requêtes d'entités dans le contrôleur, ce n'est pas le meilleur endroit pour faire celà. Les requêtes seraient bien mieux en dehors du contrôleur pour plusieurs raisons :

1. Nous serions dans l'impossibilité de réutiliser des requêtes ailleurs dans l'application sans duppliquer du code utilisant le `QueryBuilder`.
2. En dupliquant du code du `QueryBuilder`, si une requête change, il y a plusieurs modifications à faire pour répercuter le changement, ce qui est source d'erreurs.
3. En séparant la requête et le contrôleur, on devient capable de tester les requêtes indépendamment du contrôleur.

Doctrine 2 nous propose des classes de dépôt (repository) pour cela.

Les dépôts Doctrine 2

Nous avons déjà parlé des dépôts dans le chapitre précédent lorsqu'il était question de la page d'affichage des articles. Nous avons utilisé l'implémentation par défaut de la classe``DoctrineORMEntityRepository`` pour récupérer une entité du blog via ma méthode `find()`. Comme nous voulons créer une requête particulière, nous devons personnaliser un dépôt. Doctrine 2 va nous aider dans cette tâche. Mettez à jour les métadonnées de l'entité `Blog` dans le fichier `src/Blogger/BlogBundle/Entity/Blog.php`.

```
// src/Blogger/BlogBundle/Entity/Blog.php
/**
 * @ORM\Entity(repositoryClass="Blogger\BlogBundle\Repository\BlogRepository")
 * @ORM\Table(name="blog")
 * @ORM\HasLifecycleCallbacks()
 */
class Blog
{
    // ..
}
```

Vous pouvez voir que nous avons précisé l'espace de nom pour la classe `BlogRepository` associée à cette entité. Comme nous venons de mettre à jour les métadonnées de l'entité `Blog`, il faut relancer la commande `doctrine:generate:entities` comme suit :

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

Doctrine 2 va alors créer une classe vide pour notre `BlogRepository` dans `src/Blogger/BlogBundle/Repository/BlogRepository.php`.

```
<?php
// src/Blogger/BlogBundle/Repository/BlogRepository.php

namespace Blogger\BlogBundle\Repository;

use Doctrine\ORM\EntityRepository;

/**
 * BlogRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class BlogRepository extends EntityRepository
{

}
```

La classe `BlogRepository` étends la classe `EntityRepository` qui propose la méthode `find()` dont nous parlions plus tôt. Mettons à jour la classe `BlogRepository`, en déplacant le code du `QueryBuilder` du contrôleur de `Page` dedans.

```
<?php
// src/Blogger/BlogBundle/Repository/BlogRepository.php
```

```php
namespace Blogger\BlogBundle\Repository;

use Doctrine\ORM\EntityRepository;

/**
 * BlogRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class BlogRepository extends EntityRepository
{
    public function getLatestBlogs($limit = null)
    {
        $qb = $this->createQueryBuilder('b')
                   ->select('b')
                   ->addOrderBy('b.created', 'DESC');

        if (false === is_null($limit))
            $qb->setMaxResults($limit);

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

Nous avons créé la méthode `getLatestBlogs` qui va nous renvoyer les derniers articles du blog, de la même manière que le faisait le code du `QueryBuilder`. Dans la classe du repository nous avons un accès direct au `QueryBuilder` via la méthode `createQueryBuilder()`. Nous avons également ajouté un paramètre par défaut `$limit` afin de pouvoir limiter le nombre de résultats à renvoyer. Le reste ressemble beaucoup à ce qu'il y avait dans le contrôleur. Vous avez peut être remarqué que nous n'avons pas besoin de préciser quelle entité utiliser dans la méthode `from()`. C'est parce que nous sommes dans le `BlogRepository`, qui est associé à l'entité `Blog`. Si l'on regarde l'implémentation de la méthode `createQueryBuilder` de la classe `EntityRepository`, on peut voir que la méthode `from()` est appelée pour nous.

```php
// Doctrine\ORM\EntityRepository
public function createQueryBuilder($alias)
{
    return $this->_em->createQueryBuilder()
        ->select($alias)
        ->from($this->_entityName, $alias);
}
```

Mettons enfin à jour l'action `index` du contrôleur de `Page` afin de nous servir du `BlogRepository`.

```php
// src/Blogger/BlogBundle/Controller/PageController.php
class PageController extends Controller
{
    public function indexAction()
    {
        $em = $this->getDoctrine()
                   ->getEntityManager();

        $blogs = $em->getRepository('BloggerBlogBundle:Blog')
                    ->getLatestBlogs();

        return $this->render('BloggerBlogBundle:Page:index.html.twig', array(
            'blogs' => $blogs
        ));
    }

    // ..
}
```

Si vous rafraichissez la page d'accueil, rien n'aura changé : nous venons simplement de refactorer notre code, c'est à dire que nous l'avons réorganisé afin que chaque classe fasse ce qu'elle est censée faire.

Plus sur le modèle : création de l'entité de commentaire

Les articles, c'est seulement la moitié du travail quand il est question de blogguer. Nous devons également permettre aux lecteurs de les commenter. Ces commentaires doivent également être sauvegardés et liés à l'entité `Blog` car un article peut contenir plusieurs commentaires.

Nous allons commencer par poser les bases de la classe de l'entité de commentaire `Comment`. Créez un fichier dans `src/Blogger/BlogBundle/Entity/Comment.php` et collez-y le code suivant :

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

namespace Blogger\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="Blogger\BlogBundle\Repository\CommentRepository")
 * @ORM\Table(name="comment")
 * @ORM\HasLifecycleCallbacks()
 */
class Comment
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

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

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

    /**
     * @ORM\Column(type="boolean")
     */
    protected $approved;

    /**
     * @ORM\ManyToOne(targetEntity="Blog", inversedBy="comments")
     * @ORM\JoinColumn(name="blog_id", referencedColumnName="id")
     */
    protected $blog;

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

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

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

        $this->setApproved(true);
    }

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

La plupart des choses que vous voyez ici ont déjà été abordées dans le chapitre précédent, à part que nous avons utilisé les métadonnées pour faire un lien vers l'entité `Blog`. Comme un commentaire est associé à un article, nous avons créé un lien dans l'entité `Comment` vers l'entité `Blog` qui lui est associée. On fait celà en créant un lien `ManyToOne` qui cible l'entité `Blog`. On spécifie également que l'inverse de ce lien est `comments`. Pour créer cet inverse, il faut mettre à jour l'entité `Blog` afin que Doctrine 2 sache qu'un article peut contenir plusieurs commentaires. Mettez à jour l'entité `Blog` dans `src/Blogger/BlogBundle/Entity/Blog.php` pour ajouter cette association. De ce fait, nous allons pouvoir connaitre, depuis un article, quels sont les commentaires associés directement, de manière objet, et pareil pour les commentaires: nous pourrons savoir à quel article ils sont associés.

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

namespace Blogger\BlogBundle\Entity;

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

/**
 * @ORM\Entity(repositoryClass="Blogger\BlogBundle\Repository\BlogRepository")
 * @ORM\Table(name="blog")
 * @ORM\HasLifecycleCallbacks()
 */
class Blog
{
    // ..

    /**
     * @ORM\OneToMany(targetEntity="Comment", mappedBy="blog")
     */
    protected $comments;

    // ..

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

        $this->setCreated(new \DateTime());
        $this->setUpdated(new \DateTime());
    }

    // ..
}
```

Il y a plusieurs changements à noter ici. Tout d'abord, on ajoute des métadonnées au membre `$comments`. Souvenez vous que dans le chapitre précédent nous n'avons pas ajouté de métadonnées à cet attribut, car nous ne voulions pas que Doctrine 2 le fasse persister. C'est toujours vrai, mais nous voulons maintenant que Doctrine 2 remplisse ce champ avec les entités `Comment` adaptées. C'est ce que font ces métadonnées. Ensuite, Doctrine 2 a besoin que le membre `$comments` soit créé par défaut en tant qu'objet `ArrayCollection`. On fait cela dans le constructeur. Vous pouvez également noter le `use` chargé d'importer la classe `ArrayCollection`.

Comme nous venons de créer l'entité `Comment` et mis à jour l'entité `Blog`, laissons Doctrine 2 générer pour nous les accesseurs. Lancez la commande suivante :

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

Les deux entités devraient maintenant être à jour avec des accesseurs corrects. Vous allez également remarquer qu'une classe de dépôt `CommentReposity` a été créée dans `src/Blogger/BlogBundle/Repository/CommentRepository.php` comme nous l'avons précisé dans les métadonnées.

Il faut également mettre à jour la base de donnée pour répercuter les changements à nos entités. Nous pourrions utiliser `doctrine:schema:update` de la manière suivante pour cela, mais nous allons plutôt utiliser un migration Doctrine 2.

```
$ php app/console doctrine:schema:update --force
```

Les migrations Doctrine 2

L'extension et le bundle de migration Doctrine 2 n'est pas disponible de base avec la distribution standard de Symfony2, nous devons l'installer nous même comme nous l'avons fait pour les données factices. Ouvrez le fichier `deps` à la racine du projet et ajoutez l'extension comme suit :

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

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

Mettez ensuite à jour les vendors pour refléter ce changement.

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

Cela va télécharger les dernières versions de chaque dépôt sur Github et les installer au bon endroit.

> **Note**
>
> Si vous n'avez pas une machine sur laquelle Git est installée, vous allez devoir télécharger et installer vous même l'extension et le bundle.
>
> doctrine-migrations extension: Téléchargez la version actuelle depuis Github et décompressez là dans `vendor/doctrine-migrations`.
>
> DoctrineMigrationsBundle: Téléchargez la version actuelle depuis Github et décompressez là dans `vendor/bundles/Symfony/Bundle/DoctrineMigrationsBundle`.

Mettez ensuite à jour le fichier `app/autoloader.php` pour enregistrer le nouvel espace de nom. Comme ce plugin est également dans l'espace de nom `Doctrine\DBAL`, les nouveaux ajouts doivent être placés au dessus de celui déjà existant. Les espace de noms sont vérifiés de haut en bas, il faut donc les enregistrer du plus spécifique au moins spécifique.

```
// app/autoloader.php
// ...
$loader->registerNamespaces(array(
// ...
'Doctrine\\DBAL\\Migrations' => __DIR__.'/../vendor/doctrine-migrations/lib',
'Doctrine\\DBAL'             => __DIR__.'/../vendor/doctrine-dbal/lib',
// ...
));
```

Il faut maintenant enregistrer le bundle dans le noyau, situé dans `app/AppKernel.php`.

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

> **Warning**
>
> La librairie de migrations Doctrine 2 est encore actuellement en alpha. Son utilisation sur les serveurs de production est donc découragée à l'heure actuelle.

Nous sommes maintenant prêts à mettre à jour notre base de donnée pour réaliser les changements dans les entités. C'est un processus qui comprend 2 étapes: il faut tout d'abord faire découvrir à l'extension de migrations quelles ont été les changements, à travers la commande `doctrine:migrations:diff`. Il faut ensuite réaliser la migration, à partir de ces différences, à l'aide de la commande `doctrine:migrations:migrate`.

Lancez les 2 commandes qui suivent pour mettre à jour le schéma de base de donnée.

```
$ php app/console doctrine:migrations:diff
$ php app/console doctrine:migrations:migrate
```

Votre base de donnée va maintenant refléter les changements dans les entités et contenir la nouvelle table de commentaires.

> **Note**
>
> Vous pouvez également remarquer une nouvelle table appellée `migration_versions` dans votre base de données. Elle stocke les numéros de version de migrations afin que les migrations puissent savoir quel est la version actuelle de la base de donnée.

> **Tip**
>
> Doctrine 2 Migrations est un bon moyen de mettre à jour la base de donnée car les changements peuvent être faits par la programmation. Cela signifie que nous pouvons intégrer cette tâche dans un script de déploiement afin que la base de donnée soit automatiquement mise à jour lorsque l'on déploie une nouvelle version de l'application. Doctrine 2 Migrations permet également de revenir à une version précédent car charque migration propose une méthode `up` et `down`. Pour revenir à une version antérieure, il faut préciser le numéro de version vers laquelle vous souhaitez revenir en utilisant la commande suivante :

```
$ php app/console doctrine:migrations:migrate 20110806183439
```

Les données factices revisitées

Maintenant que nous avons créé l'entité `Comment`, ajoutons lui quelques données factices. C'est toujours une bonne idée de créer des données factices lorsque l'on crée une nouvelle entité. On sait qu'un commentaire doit avoir une entité `Blog` associée comme nous l'avons précisé dans les métadonnées, de ce fait lorsque l'on crée une entité `Comment` il faut lui spécifier une entité `Blog` entity. Nous avons déjà créé les données factices pour l'entité `Blog`, donc nous pourrions simplement mettre à jour le fichier qui contient ces définitions et ajouter la création des entités `Comment`. C'est peut-être OK pour le moment, mais que va-t-il se passer quand nous allons ensuite ajouter des utilisateurs, des catégories d'articles et d'autres entités à notre bundle ? Une meilleure manière de fonctionner, c'est de créer les données factices pour l'entité `Comment` dans un nouveau fichier. Un nouveau problème apparait avec cette approche : comment accéder aux entités factices de la classe `Blog` ?

Heureusement, ce problème peut aisément être résolu en créant dans références aux objets dans un des fichiers de données, référence à laquelle les autres données factices auront accès. Mettez à jour les données factices de l'entité `Blog` dans `src/Blogger/BlogBundle/DataFixtures/ORM/BlogFixtures.php` avec ce qui suit. Les changements à noter ici sont l'extension de la classe `AbstractFixture` et l'implémentation de `OrderedFixtureInterface`. Notez également les deux `use` pour importer ces classes.

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

namespace Blogger\BlogBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Blogger\BlogBundle\Entity\Blog;

class BlogFixtures extends AbstractFixture implements OrderedFixtureInterface
{
    public function load($manager)
    {
        // ..

        $manager->flush();

        $this->addReference('blog-1', $blog1);
        $this->addReference('blog-2', $blog2);
        $this->addReference('blog-3', $blog3);
        $this->addReference('blog-4', $blog4);
        $this->addReference('blog-5', $blog5);
    }

    public function getOrder()
    {
        return 1;
    }
}
```

On ajoute des références aux articles via la méthode `addReference()`. Le premier paramètre est un identifiant de référence que nous pouvons utiliser pour retrouver cet objet par la suite. Nous devons également implémenter la méthode `getOrder()` pour préciser l'ordre de chargement des données factices. Les articles doivent être chargé avant les commentaires, donc on renvoit 1.

Commentaires factices

Nous sommes maintenant prêts pour créer des données factices pour notre entité `Comment`. Créez un fichier de données factices dans `src/Blogger/BlogBundle/DataFixtures/ORM/CommentFixtures.php` et ajoutez-y le contenu suivant :

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

namespace Blogger\BlogBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Blogger\BlogBundle\Entity\Comment;
use Blogger\BlogBundle\Entity\Blog;

class CommentFixtures extends AbstractFixture implements OrderedFixtureInterface
{
    public function load($manager)
```

```
{
    $comment = new Comment();
    $comment->setUser('symfony');
    $comment->setComment('To make a long story short. You can\'t go wrong by choosing Symfony! And no one has ever been fired for using
    $comment->setBlog($manager->merge($this->getReference('blog-1')));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('David');
    $comment->setComment('To make a long story short. Choosing a framework must not be taken lightly; it is a long-term commitment. Mak
    $comment->setBlog($manager->merge($this->getReference('blog-1')));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Dade');
    $comment->setComment('Anything else, mom? You want me to mow the lawn? Oops! I forgot, New York, No grass.');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Kate');
    $comment->setComment('Are you challenging me? ');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-23 06:15:20"));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Dade');
    $comment->setComment('Name your stakes.');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-23 06:18:35"));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Kate');
    $comment->setComment('If I win, you become my slave.');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-23 06:22:53"));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Dade');
    $comment->setComment('Your SLAVE?');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-23 06:25:15"));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Kate');
    $comment->setComment('You wish! You\'ll do shitwork, scan, crack copyrights...');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-23 06:46:08"));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Dade');
    $comment->setComment('And if I win?');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-23 10:22:46"));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Kate');
    $comment->setComment('Make it my first-born!');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-23 11:08:08"));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Dade');
    $comment->setComment('Make it our first-date!');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-24 18:56:01"));
    $manager->persist($comment);

    $comment = new Comment();
    $comment->setUser('Kate');
    $comment->setComment('I don\'t DO dates. But I don\'t lose either, so you\'re on!');
    $comment->setBlog($manager->merge($this->getReference('blog-2')));
    $comment->setCreated(new \DateTime("2011-07-25 22:28:42"));
```

```
        $manager->persist($comment);

        $comment = new Comment();
        $comment->setUser('Stanley');
        $comment->setComment('It\'s not gonna end like this.');
        $comment->setBlog($manager->merge($this->getReference('blog-3')));
        $manager->persist($comment);

        $comment = new Comment();
        $comment->setUser('Gabriel');
        $comment->setComment('Oh, come on, Stan. Not everything ends the way you think it should. Besides, audiences love happy endings.');
        $comment->setBlog($manager->merge($this->getReference('blog-3')));
        $manager->persist($comment);

        $comment = new Comment();
        $comment->setUser('Mile');
        $comment->setComment('Doesn\'t Bill Gates have something like that?');
        $comment->setBlog($manager->merge($this->getReference('blog-5')));
        $manager->persist($comment);

        $comment = new Comment();
        $comment->setUser('Gary');
        $comment->setComment('Bill Who?');
        $comment->setBlog($manager->merge($this->getReference('blog-5')));
        $manager->persist($comment);

        $manager->flush();
    }

    public function getOrder()
    {
        return 2;
    }
}
```

Comme nous l'avons fait dans la classe `BlogFixtures`, la classe `CommentFixtures` étend elle aussi la classe `AbstractFixture` et implémente `OrderedFixtureInterface`. Cela signifie que nous devons également implémenter la méthode `getOrder()`. Cette fois-ci, la valeur de retour est 2, ce qui nous assure que ces informations seront chargées après celles des articles.

On peut également voir comment les références aux entités `Blog`, que nous avions créées précédemment, ont été utilisées.

```
$comment->setBlog($manager->merge($this->getReference('blog-2')));
```

Nous sommes maintenant prêt à charger ces données dans la base de données :

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

Affichage des commentaires :

On peut maintenant afficher les commentaires associés à chaque article du blog. Commençons par mettre à jour le `CommentReposity` avec une méthode pour charger les derniers commentaires validés d'un article.

Dépôt de commentaires

Ouvrez la classe `CommentRepository` dans `src/Blogger/BlogBundle/Repository/CommentRepository.php` et remplacez son contenu par ce qui suit :

```
<?php
// src/Blogger/BlogBundle/Repository/CommentRepository.php

namespace Blogger\BlogBundle\Repository;

use Doctrine\ORM\EntityRepository;

/**
 * CommentRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class CommentRepository extends EntityRepository
{
    public function getCommentsForBlog($blogId, $approved = true)
    {
```

```
        $qb = $this->createQueryBuilder('c')
                    ->select('c')
                    ->where('c.blog = :blog_id')
                    ->addOrderBy('c.created')
                    ->setParameter('blog_id', $blogId);

        if (false === is_null($approved))
            $qb->andWhere('c.approved = :approved')
               ->setParameter('approved', $approved);

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

La méthode que nous venons d'ajouter récupère les commentaires associés à un article. Pour cela, on ajoute une clause `where` à notre requête, qui utilise un paramètre nommé, paramètre qui est associé à une variable grâce à la méthode `setParameter()`. Vous devriez toujours utiliser des paramètres plutôt que de spécifier les valeurs directement comme ceci :

```
->where('c.blog = ' . blogId)
```

En effet dans cet exemple, la valeur de `$blogId` n'a pas été assainie, ce qui pourrait mener à des failles de sécurité en laissant la porte ouverte à des attaques par injection SQL.

Contrôleur des articles

Il faut maintenant mettre à jour l'action `show` du contrôleur `Blog` pour récupérer les commentaires de l'article. Mettez à jour le contrôleur de `Blog` dans `src/Blogger/BlogBundle/Controller/BlogController.php` avec le contenu suivant :

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

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

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

    $comments = $em->getRepository('BloggerBlogBundle:Comment')
                   ->getCommentsForBlog($blog->getId());

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

Nous utilisons la nouvelle méthode du `CommentReposity` pour récupérer les commentaires validés de l'article. La collection `$comments` est également passée en paramètre du template.

Template de l'affichage des articles

Maintenant que nous avons une liste des commentaires de l'article, nous pouvons mettre à jour la page d'affichage des articles afin de les y afficher. Nous pourrions simplement placer l'affichage des commentaires directement dans le template d'affichage des articles, mais comme les commentaires sont une entité propre, c'est mieux de séparer leur affichage dans un template séparé, que l'on inclut dans un autre. Cela nous permet de réutiliser l'affichage des commentaires ailleurs dans l'application. Mettez à jour le template d'affichage des articles dans `src/Blogger/BlogBundle/Resources/public/views/Blog/show.html.twig` avec ce qui suit :

```
{# src/Blogger/BlogBundle/Resources/public/views/Blog/show.html.twig #}

{# .. #}

{% block body %}
    {# .. #}

    <section class="comments" id="comments">
        <section class="previous-comments">
            <h3>Comments</h3>
            {% include 'BloggerBlogBundle:Comment:index.html.twig' with { 'comments': comments } %}
        </section>
```

```
    </section>
{% endblock %}
```

Vous pouvez remarquer l'utilisation d'un nouveau tag Twig, le tag `include`. Comme son nom l'indique, il inclut le contenu du template fourni en paramètres, ici `BloggerBlogBundle:Comment:index.html.twig`. On peut également lui passer des arguments. Dans le cas présent, on lui fournit une collection d'entités `Comment` à afficher.

Template d'affichage des commentaires

Le template `BloggerBlogBundle:Comment:index.html.twig` que l'on inclut plus haut n'existe pas pour le moment et il faut le créer. Comme il s'agit simplement d'un template qui sera inclut dans un autre, pas besoin de créer une route ou un contrôleur pour cela : il suffit de créer un fichier. Créez un nouveau fichier dans `src/Blogger/BlogBundle/Resources/public/views/Comment/index.html.twig` et collez-y ce qui suit :

```
{# src/Blogger/BlogBundle/Resources/public/views/Comment/index.html.twig #}

{% for comment in comments %}
    <article class="comment {{ cycle(['odd', 'even'], loop.index0) }}" id="comment-{{ comment.id }}">
        <header>
            <p><span class="highlight">{{ comment.user }}</span> commented <time datetime="{{ comment.created|date('c') }}">{{ comment.crea
        </header>
        <p>{{ comment.comment }}</p>
    </article>
{% else %}
    <p>There are no comments for this post. Be the first to comment...</p>
{% endfor %}
```

Comme vous pouvez le voir, on traverse la collection d'entité `Comment` et affiche les commentaires. On peut également voir une des fonctions sympa de Twig, la fonction `cycle`. Cette fonction avance en boucle d'une case à travers les valeurs du tableau à chaque itération. L'indice de boucle courant est obtenu grâce à la variable spéciale `loop.index0`, qui garde le compte des itérations dans la boucle, en partant de 0. Il y a plusieurs autres variables particulières disponibles lorsque l'on est à l'intérieur d'une boucle. Vous pouvez également remarquer la présence d'un identifiant HTML dans l'élément `article`. Cela nous permettra par la suite de créer des permalinks (liens permanents) vers les commentaires.

CSS d'affichage des commentaires

Ajoutons également un peu de CSS pour que les commentaires soient agréables à regarder. Mettez à jour la feuille de style dans `src/Blogger/BlogBundle/Resorces/public/css/blog.css` en y ajoutant ce qui suit :

```
/** src/Blogger/BlogBundle/Resorces/public/css/blog.css **/
.comments { clear: both; }
.comments .odd { background: #eee; }
.comments .comment { padding: 20px; }
.comments .comment p { margin-bottom: 0; }
.comments h3 { background: #eee; padding: 10px; font-size: 20px; margin-bottom: 20px; clear: both; }
.comments .previous-comments { margin-bottom: 20px; }
```

Si vous jetez maintenant un oeil à la page d'affichage des articles, par exemple `http://symblog.dev/app_dev.php/2`, vous pouvez voir l'affichage des commentaires d'articles.

Affichage des commentaires d'articles dans Symblog

Ajouter des commentaires

La dernière partie de ce chapitre va nous faire ajouter aux utilisateurs la possibilité de commenter les articles. Cela va être possible grâce à un formulaire sur la page d'affichage des articles. Nous avons déjà parlé de la création de formulaires dans Symfony2 lorsque nous avons créé la page de contacts. Plutôt que créer le formulaire nous même, nous allons utiliser Symfony2 pour faire cela pour nous.

Lancez la commande suivante pour générer la classe `CommentType` pour l'entité `Comment`.

```
$ php app/console generate:doctrine:form BloggerBlogBundle:Comment
```

Vous remarquerez à nouveau ici l'utilisation ici de la version raccourcie pour spécifier l'entité `Comment`.

> **Tip**
>
> Vous avez peut être remarqué que la commande `doctrine:generate:form` est également disponible. Elle fait la même chose, mais l'espace de nom utilisé est différent.

Cette commande a généré pour nous la classe `CommentType` dans `src/Blogger/BlogBundle/Form/CommentType.php`.

```php
<?php
// src/Blogger/BlogBundle/Form/CommentType.php

namespace Blogger\BlogBundle\Form;

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

class CommentType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
            ->add('user')
            ->add('comment')
            ->add('approved')
            ->add('created')
            ->add('updated')
            ->add('blog')
        ;
    }

    public function getName()
    {
        return 'blogger_blogbundle_commenttype';
    }
}
```

Nous avons déjà exploré ce qui se passe ici pour la classe `EnquiryType`. Nous pourrions commencer par personnaliser cette classe tout de suite, mais nous allons d'abord afficher le formulaire.

Afficher le formulaire de commentaires.

Comme nous voulons que l'utilisateur puisse ajouter des commentaires depuis la page d'affichage des articles, nous pourrions créer le formulaire dans l'action `show` du contrôleur `Blog` et afficher le formulaire directement dans le template `show`. Il est toutefois mieux de séparer ce code, comme nous l'avions fait pour l'affichage des commentaires. La différence entre afficher les commentaires et afficher le formulaire de commentaires, c'est que cette seconde tâche doit être traitée, elle nécessite donc un contrôleur. Cela nous amènera donc à procéder légèrement différement de ce que nous venons de faire, où il était simplement question d'inclure un template.

Routage

Nous devons créer une nouvelle route pour gérer le traitement du formulaire soumis. Ajoutez une nouvelle route dans le fichier `src/Blogger/BlogBundle/Resources/config/routing.yml`.

```yaml
BloggerBlogBundle_comment_create:
    pattern:  /comment/{blog_id}
    defaults: { _controller: BloggerBlogBundle:Comment:create }
    requirements:
        _method:  POST
        blog_id: \d+
```

Le contrôleur

Ensuite, il faut créer le nouveau contrôleur `Comment` auquel nous faisons référence juste au dessus. Créez un fichier dans `src/Blogger/BlogBundle/Controller/CommentController.php` et collez-y le code qui suit :

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

namespace Blogger\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Blogger\BlogBundle\Entity\Comment;
use Blogger\BlogBundle\Form\CommentType;

/**
 * Comment controller.
 */
class CommentController extends Controller
```

```
{
    public function newAction($blog_id)
    {
        $blog = $this->getBlog($blog_id);

        $comment = new Comment();
        $comment->setBlog($blog);
        $form    = $this->createForm(new CommentType(), $comment);

        return $this->render('BloggerBlogBundle:Comment:form.html.twig', array(
            'comment' => $comment,
            'form'    => $form->createView()
        ));
    }

    public function createAction($blog_id)
    {
        $blog = $this->getBlog($blog_id);

        $comment  = new Comment();
        $comment->setBlog($blog);
        $request = $this->getRequest();
        $form    = $this->createForm(new CommentType(), $comment);
        $form->bindRequest($request);

        if ($form->isValid()) {
            // TODO: Persist the comment entity

            return $this->redirect($this->generateUrl('BloggerBlogBundle_blog_show', array(
                'id' => $comment->getBlog()->getId())) .
                '#comment-' . $comment->getId()
            );
        }

        return $this->render('BloggerBlogBundle:Comment:create.html.twig', array(
            'comment' => $comment,
            'form'    => $form->createView()
        ));
    }

    protected function getBlog($blog_id)
    {
        $em = $this->getDoctrine()
                   ->getEntityManager();

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

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

        return $blog;
    }

}
```

On crée 2 actions dans le contrôleur `Comment`, une pour l'action `new` et une pour l'action `create`. L'action `new` est chargée d'afficher le formulaire de commentaires, alors que l'action `create` a pour mission de traiter le formulaire de commentaire soumis. Le bout de code a l'air imposant, mais il n'y a rien de nouveau ici, tout a déjà été abordé dans le chapitre 2 lorsque l'on a créé le formulaire de contact. Avant d'avancer, assurez vous toutefois d'avoir bien compris ce qui se passe dans ce contrôleur.

Validation du formulaire

Nous ne voulons pas que les utilisateurs puissent proposer des commentaires avec des valeurs vides pour le nom d'utilisateur ou le contenu. Pour cela, nous allons retourner dans les validateurs dont nous avions déjà parlé au chapitre 2, lors de la soumission du formulaire de contact. Mettez à jour l'entité `Comment` dans `src/Blogger/BlogBundle/Entity/Comment.php` avec ce qui suit.

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

// ..

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

// ..
class Comment
```

```
{
    // ..

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('user', new NotBlank(array(
            'message' => 'You must enter your name'
        )));
        $metadata->addPropertyConstraint('comment', new NotBlank(array(
            'message' => 'You must enter a comment'
        )));
    }

    // ..
}
```

Ces contraintes vérifient que le nom d'utilisateur et le contenu du commentaire ne sont pas vides. Nous avons également ajouté l'option `message` aux deux contraintes pour remplacer le message par défaut. N'oubliez pas d'ajouter l'espace de nom pour `ClassMetadata` et `NotBlank` comme c'est le cas ici.

La vue

Il faut ensuite créer les 2 templates pour les actions `new` et `create`. Commencez par créer un nouveau fichier dans `src/Blogger/BlogBundle/Resources/public/views/Comment/form.html.twig` et collez-y le code qui suit :

```
{# src/Blogger/BlogBundle/Resources/public/views/Comment/form.html.twig #}

<form action="{{ path('BloggerBlogBundle_comment_create', { 'blog_id' : comment.blog.id } ) }}" method="post" {{ form_enctype(form) }} clas
    {{ form_widget(form) }}
    <p>
        <input type="submit" value="Submit">
    </p>
</form>
```

Le but de ce template est simple, il affiche simplement le formulaire de commentaires. Vous pourrez également remarquer que l'`action` du formulaire est à `POST` vers la route que nous venons de créer, `BloggerBlogBundle_comment_create`.

Maintenant, ajoutons le template pour la vue `create`. Créez un nouveau fichier dans `src/Blogger/BlogBundle/Resources/public/views/Comment/create.html.twig` et collez-y le code suivant :

```
{% extends 'BloggerBlogBundle::layout.html.twig' %}

{% block title %}Add Comment{% endblock%}

{% block body %}
    <h1>Add comment for blog post "{{ comment.blog.title }}"</h1>
    {% include 'BloggerBlogBundle:Comment:form.html.twig' with { 'form': form } %}
{% endblock %}
```

Comme l'action `create` du contrôleur `Comment` s'occupe de traiter le formulaire, il doit également permettre de l'afficher, car il pourrait y avoir des erreurs dans le formulaire. Nous utilisons à nouveau `BloggerBlogBundle:Comment:form.html.twig` pour afficher le formulaire et ainsi éviter la duplication de code.

Maintenant, mettons à jour le template d'affichage des articles pour afficher le formulaire d'ajout de commentaires. Mettez à jour le template dans `src/Blogger/BlogBundle/Resources/public/views/Blog/show.html.twig` avec ce qui suit :

```
{# src/Blogger/BlogBundle/Resources/public/views/Blog/show.html.twig #}

{# .. #}

{% block body %}

    {# .. #}

    <section class="comments" id="comments">
        {# .. #}

        <h3>Add Comment</h3>
        {% render 'BloggerBlogBundle:Comment:new' with { 'blog_id': blog.id } %}
    </section>
{% endblock %}
```

Nous utilisons un nouveau tag Twig ici, le tag `render`. Ce tag permet d'afficher le contenu d'un contrôleur dans un template. Dans le cas présent, il affiche le contenu de l'action `BloggerBlogBundle:Comment:new`, en lui fournissant le paramètre nécessaires, l'identifiant de l'article.

Si vous regardez maintenant une des pages d'affichage des articles, tel que `http://symblog.dev/app_dev.php/2`, vous allez voir qu'une exception Symfony2 est lancée.

Exception Symfony 2 toString()

Cette exception est lancée par le template `BloggerBlogBundle:Blog:show.html.twig`. Si l'on regarde à la ligne 25 du fichier `BloggerBlogBundle:Blog:show.html.twig`, on peut voir que cette ligne crée une erreur au moment d'embarquer le contrôleur `BloggerBlogBundle:Comment:create`.

```
{% render 'BloggerBlogBundle:Comment:create' with { 'blog_id': blog.id } %}
```

En regardant un peu plus attentivement cette exception, on peut voir qu'elle nous donne plus d'informations sur la raison pour laquelle l'exception a été lancée.

> Entities passed to the choice field must have a "__toString()" method defined

Cela nous dit qu'un champ de choix que l'on essaye d'afficher n'a pas de méthode `__toString()` dans l'entité à laquelle il est associé. Un champ de choix est un élément de formulaire qui fournit à l'utilisateur plusieurs choix, tel qu'un élément `select` (une liste déroulante). Vous vous demandez sûrement où est-ce que l'on affiche un champ de choix dans le formulaire de commentaires... Si vous regardez à nouveau le template du formulaire de commentaires, vous pouvez voir que l'on affiche le formulaire grâce à la fonction Twig `{{ form_widget(form) }}`. Cette fonction affiche le formulaire entité de manière basique. Retournons donc dans la classe qui crée ce formulaire, la classe `CommentType`. On peut voir que plusieurs champs sont ajoutés au formulaire via l'objet `FormBuilder`. On ajoute en particulier un champs `blog`. Si vous vous souvenez du chapitre 2, nous avions parlé de comment le `FormBuilder` essaye de deviner le type de champ à afficher à partir des métadonnées qui lui sont associées. Comme nous avons établi un lien entre les entités `Comment` et `Blog`, le `FormBuilder` a deviné que le commentaire devait avoir un champ de choix, afin de permettre à l'utilisateur de préciser l'article auquel il est associé. C'est pourquoi nous avons un champ de choix dans le formulaire, et pourquoi Symfony2 lance une exception. On peut résoudre le problème en ajoutant la méthode `__toString()` dans l'entité `Blog`.

```
// src/Blogger/BlogBundle/Entity/Blog.php
public function __toString()
{
    return $this->getTitle();
}
```

> **Tip**
>
> Les messages d'erreur de Symfony2 fournissent beaucoup d'informations pour décrire les problèmes qui viennent d'apparaitre. Lisez toujours les messages d'erreur car ils facilitent grandement le processus de débug. Les messages d'erreur fournissent également une trace complète de la pile d'appel pour que vous voyiez les étapes qui ont mené à cette erreur.

Maintenant, lorsque vous rafraichissez la page vous devriez voir l'affichage du formulaire de commentaires. Vous pourrez également remarquer l'affichage indésirable de certains champs, tel que `approved`, `created`, `updated` et `blog`. C'est parce que nous n'avons pas personnalisé précédemment la classe `CommentType`, générée automatiquement.

> **Tip**
>
> Les champs affichés semblent tous être affichés avec un type de champ adapté : le champ `user` est de type `text`, le champ `comment``est une ``textarea`, les 2 champs `DateTime` proposent plusieurs champs `select` qui permettent de sélectionner une date, etc.
>
> C'est parce que le `FormBuilder` est capable de deviner le type d'élément de formulaire associé à un élément de l'entité. Il est capable de faire cela à partir des métadonnées que l'on lui fournit. Comme nous avons précisé des métadonnées assez spécifiques pour l'entité `Comment`, le `FormBuilder` est capable de deviner de manière précise les bons types de champ à afficher.

Mettons maintenant à jour le fichier `src/Blogger/BlogBundle/Form/CommentType.php` afin de n'afficher que les champ dont nous avons besoin.

```
<?php
// src/Blogger/BlogBundle/Form/CommentType.php

// ..
class CommentType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
```

```
            ->add('user')
            ->add('comment')
        ;
    }

    // ..
}
```

Si vous rafraichissez maintenant la page, seul les champs pour le nom d'utilisateur et pour le corps du commentaire sont affichés. Si vous vouliez soumettre le formulaire maintenant, le commentaire ne serait pas sauvegardé dans la base de données, car le contrôleur ne fait rien de l'entité `Comment` si elle passe la validation. Nous avons déjà vu comment persister des éléments dans la base de données lors de la création des données factices, nous allons faire la même chose ici avec les commentaires. Mettez à jour l'action `create` du contrôleur `Comment` afin de persister les entités `Comment` dans la base de données.

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

// ..
class CommentController extends Controller
{
    public function createAction($blog_id)
    {
        // ..

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

            return $this->redirect($this->generateUrl('BloggerBlogBundle_blog_show', array(
                'id' => $comment->getBlog()->getId())) .
                '#comment-' . $comment->getId()
            );
        }

        // ..
    }
}
```

Persister les entités `Comment` nécessite simplement d'appeler `persist()` et `flush()`. Souvenez vous que les formulaires traitent seulement avec des objets PHP, et que Doctrine 2 gère et sauve ces objets. Il n'y a pas de lien direct entre la soumission d'un formulaire et les données soumises qui sont envoyées dans la base de données.

Vous devriez maintenant pouvoir ajouter des commentaires aux articles.

Ajout de commentaire aux articles dans Symblog

Conclusion

Nous avons bien progressé dans ce chapitre. Notre site de blogging commence à avoir les fonctionnalités que l'on est en droit d'attendre de lui. Nous avons maintenant les bases de la page d'accueil et de l'entité commentaire. Les utilisateurs peuvent poster des commentaires dans les articles et lire ceux des autres utilisateurs. Nous avons vu comment créer des données factices qui sont référencées entre plusieurs fichiers de données factices, et utilisé les migrations Doctrine 2 pour conserver une trace des changements dans le schéma de base de donnée.

Dans la prochaine partie, nous allons construire la barre latérale pour inclure le nuage de tags et les commentaires récents. Nous allons également étendre Twig en créant nos propres filtres. Nous regarderons enfin comment utiliser la librairie Assetic pour nous aider dans la gestion de fichiers externes.