symblog

creating a blog in Symfony2

[Part 4] - The Comments Model: Adding comments, Doctrine Repositories and Migrations

Overview

This chapter will build on the blog model we defined in the previous chapter. We will create the comment model, which will handle comments for blog posts. We will be introduced to creating relationships between models, as a blog post can contain many comments. We will use the Doctrine 2 QueryBuilder and Doctrine 2 Repository classes to retrieve entities from the database. The concept of Doctrine 2 Migrations will also be explored which provide a programmatic way to deploy changes to the database. At the end of this chapter you will have created the comment model and linked it together with the blog model. We will also have created the homepage, and provided the ability for users to submit comments for a blog post.

The Homepage

We will begin this chapter by building the homepage. In true blogger fashion it will display snippets of each blog post, ordered from newest to oldest. The full blog post will be available via links to the blog show page. As we have already built the route, controller and view for the homepage we can simply update these.

Retrieving the blogs: Querying the model

In order to display the blogs, we need to retrieve them from the database. Doctrine 2 provides the **Doctrine Query Language** (DQL) and a **QueryBuilder** to achieve this (You can also run raw SQL through Doctrine 2 but this method is discouraged as it takes away the database abstraction Doctrine 2 gives us). We will use the `QueryBuilder` as it provides a nice object oriented way for us to generate DQL, which we can use to query the database. Let’s update the `index` action of the`Page` controller located at `src/Blogger/BlogBundle/Controller/PageController.php` to pull the blogs from the database.

```
// 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
        ));
    }

    // ..
}
```

We begin by getting an instance of the `QueryBuilder` from the `EntityManager`. This allows us to start constructing the query using the many methods the `QueryBuilder` provides. The full list of available methods is available via the `QueryBuilder` documentation. A good place to start is with the **helper methods**. These are the methods we use, such as `select()`, `from()` and `addOrderBy()`. As with previous interactions with Doctrine 2, we can use the short hand notation to reference the `Blog` entity via`BloggerBlogBundle:Blog` (remember this is the same as doing `Blogger\BlogBundle\Entity\Blog`). When we have finished specifying the criteria for the query, we call the `getQuery()` method which returns a `DQL` instance. We are not able to get results from the`QueryBuilder` object, we must always convert this to a `DQL` instance first. The `DQL` instance provides the `getResult()` method that returns a collection of `Blog` entities. We will see later that the `DQL` instance has a **number of methods** for returning results including `getSingleResult()` and `getArrayResult()`.

The View

Now we have a collection of `Blog` entities we need to display them. Replace the content of the homepage template located at `src/Blogger/BlogBundle/Resources/views/Page/index.html.twig` with the following.

```
{# 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 %}
```

We introduce one of the Twig control structures here, the `for..else..endfor` structure. If you have not used a templating engine before you are probably familiar with the following PHP snippet.

```
<?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 ?>
```

The Twig `for..else..endfor` control structure is a much cleaner way of achieving this task. Most of the code within the homepage template is concerned with outputting the blog information in HTML. However, there are a few things we need to note. Firstly, we make use of the Twig `path` function to generate the routes for the blog show page. As the blog show page requires a blog `ID` to be present in the URL, we need to pass this as an argument into the `path` function. This can be seen with the following.

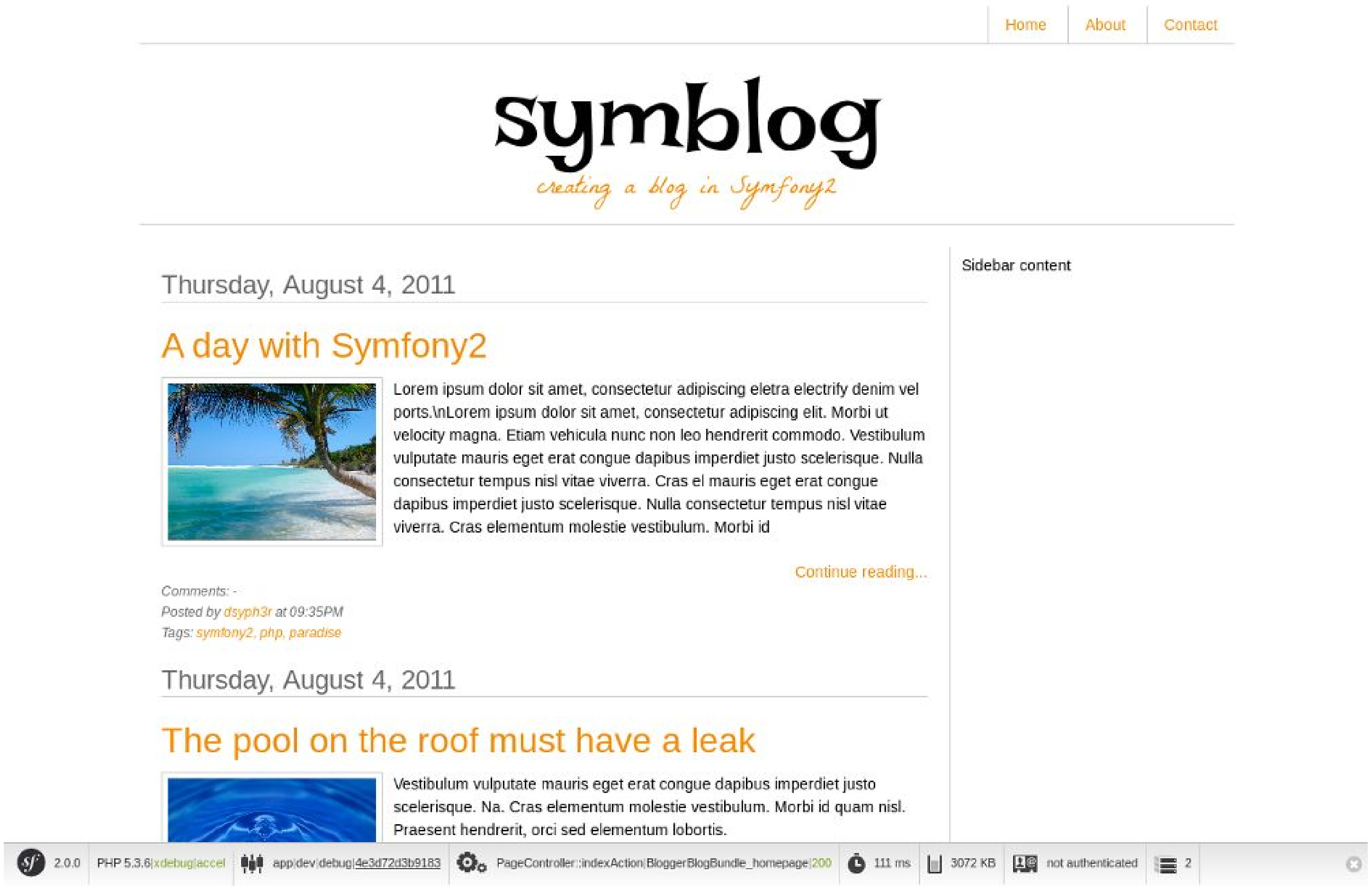
```
<h2><a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id }) }}">{{ blog.title }}</a></h2>
```

Secondly we output the blog content using `<p>{{ blog.blog(500) }}</p>`. The `500` argument we pass in, is the max length of the blog post we want to receive back from the function. For this to work we need to update the `getBlog` method that Doctrine 2 generated for us previously. Update the `getBlog` method in the `Blog` entity located at `src/Blogger/BlogBundle/Entity/Blog.php`.

```
// 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;
}
```

As the usual behavior of the `getBlog` method should be to return the entire blog post, we set the `$length` parameter to have a default of `null`. If `null` is passed in, the entire blog post is returned.

Now if you point your browser to `http://symblog.dev/app_dev.php/` you should see the homepage displaying the latest blog post entries. You should also be able to navigate to the full blog post for each entry by clicking the blog title or the 'continue reading...' link.


Home About Contact

symblog

creating a blog in Symfony2

Thursday, August 4, 2011

A day with Symfony2

Lorem ipsum dolor sit amet, consectetur adipiscing eletra electrify denim vel ports.\nLorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi ut velocity magna. Etiam vehicula nunc non leo hendrerit commodo. Vestibulum vulputate mauris eget erat congue dapibus imperdiet justo scelerisque. Nulla consectetur tempus nisl vitae viverra. Cras el mauris eget erat congue dapibus imperdiet justo scelerisque. Nulla consectetur tempus nisl vitae viverra. Cras elementum molestie vestibulum. Morbi id

Continue reading...

Comments: -
Posted by dsyph3r at 09:35PM
Tags: symfony2, php, paradise

Thursday, August 4, 2011

The pool on the roof must have a leak

Vestibulum vulputate mauris eget erat congue dapibus imperdiet justo scelerisque. Na. Cras elementum molestie vestibulum. Morbi id quam nisl. Praesent hendrerit, orci sed elementum lobortis.

Sidebar content

2.0.0 | PHP 5.3.6|xdebug|accel | app|dev|debug|4e3d72d3b9183 | PageController::indexAction|BloggerBlogBundle_homepage|200 | 111 ms | 3072 KB | not authenticated | 2

While we can query for entities in the controller, it is not the best place to. The querying would be better placed outside of the controller for a number of reasons:

1. We would be unable to reuse the query elsewhere in the application, without duplicating the `QueryBuilder`code.
2. If we did duplicate the `QueryBuilder` code, we would have to make multiple modifications in the future if the query needed changing.
3. Separating the query and the controller would allow us to test the query independently of the controller.

Doctrine 2 provides the Repository classes to facilitate this.

Doctrine 2 Repositories

We have already been introduced to the Doctrine 2 Repository classes in the previous chapter when we created the blog show page. We used the default implementation of the `Doctrine\ORM\EntityRepository` class to retrieve a blog entity from the database via the `find()` method. As we want to create a custom query, we need to create a custom repository. Doctrine 2 is able to assist in this task. Update the `Blog` entity metadata located in the file at `src/Blogger/BlogBundle/Entity/Blog.php`.

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

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

You can see we have specified the namespace location for the `BlogRepository` class that this entity is associated with. As we have updated the Doctrine 2 metadata for the `Blog` entity, we need to re-run the `doctrine:generate:entities` task as follows.

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

Doctrine 2 will have created the shell class for the `BlogRepository` located at `src/Blogger/BlogBundle/Entity/Repository/BlogRepository.php`.

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

namespace Blogger\BlogBundle\Entity\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
{

}
```

The `BlogRepository` class extends the `EntityRepository` class which provides the `find()` method we used earlier. Let’s update the`BlogRepository` class, moving the `QueryBuilder` code from the `Page` controller into it.

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

namespace Blogger\BlogBundle\Entity\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();
    }
}
```

We have created the method `getLatestBlogs` which will return the latest blog entries, much in the same way the controller`QueryBuilder` code did. In the repository class we have direct access to the `QueryBuilder` via the `createQueryBuilder()` method. We have also added a default `$limit` parameter so we can limit the number of results to return. The result of the query is much the same as it was in the controller. You may have noticed that we did not need to specify the entity to use via the `from()`method. That’s because we are within the `BlogRepository` which is associated with the `Blog` entity. If we look at the implementation of the `createQueryBuilder` method in the `EntityRepository` class we can see the `from()` method is called for us.

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

Finally let’s update the `Page` controller `index` action to use the `BlogRepository`.

```
// 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
        ));
    }

    // ..
}
```

Now when you refresh the homepage it should display exactly the same as before. All we have done is refactored our code so the correct classes are performing the correct tasks.

More on the Model: Creating the Comment Entity

Blogs are only half the story when it comes to blogging. We also need to allow readers the ability to comment on blog posts. These comments also need to be persisted, and linked to the `Blog` entity as a blog can contain many comments.

We will start by defining the basics of the `Comment` Entity class. Create a new file located at `src/Blogger/BlogBundle/Entity/Comment.php`and paste in the following.

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

namespace Blogger\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="Blogger\BlogBundle\Entity\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());
    }
}
```

Most of what you see here, we have already covered in the previous chapter, however we have used metadata to set up a link to the `Blog` entity. As a comment is for a blog, we have setup a link in the `Comment` entity to the `Blog` entity it belongs to. We do this by specify a `ManyToOne` link targeting the `Blog` entity. We also specify that the inverse of this link will be available via `comments`. To create this inverse, we need to update the `Blog` entity so Doctrine 2 knows that a blog can contain many comments. Update the `Blog` entity located at `src/Blogger/BlogBundle/Entity/Blog.php` to add this mapping.

```
<?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\Entity\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());
    }

    // ..
}
```

There are a few changes to point out here. First we add metadata to the `$comments` member. Remember in the previous chapter we didn't add any metadata for this member because we didn't want Doctrine 2 to persist it. This is still true, however, we do want Doctrine 2 to be able to populate this member with the relevant `Comment` entities. That is what the metadata achieves. Secondly, Doctrine 2 requires that we default the `$comments` member to an `ArrayCollection` object. We do this in the `constructor`. Also, note the `use` statement to import the `ArrayCollection` class.

As we have now created the `Comment` entity, and updated the `Blog` entity, let's get Doctrine 2 to generate the accessors. Run the following Doctrine 2 task as before to achieve this.

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

Both entities should now be up-to-date with the correct accessor methods. You will also notice the `CommentRepository` class has been created at `src/Blogger/BlogBundle/Entity/Repository/CommentRepository.php` as we specified this in the metadata.

Finally we need to update the database to reflect the changes to our entities. We could use the `doctrine:schema:update` task as follows to do this, but instead we will introduce Doctrine 2 Migrations.

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

Doctrine 2 Migrations

The Doctrine 2 Migrations extension and bundle do not come with the Symfony2 Standard Distribution, we need to manually install them as we did with the Data Fixtures extension and bundle. Open up the `composer.json` file located in the project root and add the Doctrine 2 Migrations extension and bundle to it as follows.

```
"require": {
    // ...
    "doctrine/doctrine-migrations-bundle": "dev-master",
    "doctrine/migrations": "dev-master"
}
```

Next update the vendors to reflect these changes.

```
$ php composer.phar update
```

This will pull down the latest version of each of the repositories from GitHub and install them to the required locations.

Now let's register the bundle in the kernel located at `app/AppKernel.php`.

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

We are now ready to update the database to reflect the entity changes. This is a 2 step process. First we need to get Doctrine 2 Migrations to work out the differences between the entities and the current database schema. This is done with the `doctrine:migrations:diff` task. Secondly we need to actually perform the migration based on the previously created diff. This is done with the `doctrine:migrations:migrate` task.

Run the following 2 commands to update the database schema.

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

Your database will now reflect the latest entity changes and contain the new comment table.

Note

You will also notice a new table in the database called `migration_versions`. This stores the migration version numbers so the migration task is able to see what the current version of the database is.

Tip

Doctrine 2 Migrations are a great way to update the production database as the changes can be done programatically. This means we can integrate this task into a deployment script so the database is updated automatically when we deploy a new release of the application. Doctrine 2 Migrations also allow us to roll back the changes as every created Migration has a `up` and `down` method. To roll back to a previous version you need to specify the version number you would like to roll back to using the following task.

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

Data Fixtures: Revisited

Now we have the `Comment` entity created, let's add some fixtures for it. It is always a good idea to add some fixtures each time you create an entity. We know that a comment must have a related `Blog` entity set as it is setup this way in the metadata, therefore when creating fixtures for `Comment` entities we will need to specify the `Blog` entity. We have already created the fixtures for the `Blog` entity so we could simply update this file to add the `Comment` entities. This may be manageable for now, but what happens when we later add users, blog categories, and a whole load of other entities to our bundle? A better way would be to create a new file for the `Comment` entity fixtures. The problem with this approach is how do we access the `Blog` entities from the blog fixtures.

Fortunately this can be achieved easily by setting references to objects in one fixture file that other fixtures can access. Update the `Blog` entity `DataFixtures` located at `src/Blogger/BlogBundle/DataFixtures/ORM/BlogFixtures.php` with the following. The changes to note here are the extension of the `AbstractFixture` class and the implementation of the `OrderedFixtureInterface`. Also note the 2 new use statements to import these classes.

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

namespace Blogger\BlogBundle\DataFixtures\ORM;

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

class BlogFixtures extends AbstractFixture implements OrderedFixtureInterface
{
    public function load(ObjectManager $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;
    }
}
```

We add references to the blog entities using the `addReference()` method. This first parameter is a reference identifier we can use the retrieve the object later. Finally we must implement the `getOrder()` method to specify the loading order of the fixtures. Blogs must be loaded before comments so we return 1.

Comment Fixtures

We are now ready to define some fixtures for our `Comment` entity. Create a fixtures file at`src/Blogger/BlogBundle/DataFixtures/ORM/CommentFixtures.php` and add the following content:

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

namespace Blogger\BlogBundle\DataFixtures\ORM;

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

class CommentFixtures extends AbstractFixture implements OrderedFixtureInterface
{
    public function load(ObjectManager $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 Symfony.');
        $comment->setBlog($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. Make sure that you make the right selection!');
        $comment->setBlog($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;
    }
}
```

As with the modifications we made the `BlogFixtures` class, the `CommentFixtures` class also extends the `AbstractFixture` class and implements the `OrderedFixtureInterface`. This means we must also implement the `getOrder()` method. This time we set the return value to 2, ensuring these fixtures will be loaded after the blog fixtures.

We can also see how the references to the `Blog` entities we created earlier are being used.

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

We are now ready to load the fixtures into the database.

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

Displaying Comments

We can now display the comments related to each blog post. We begin by updating the `CommentRepository` with a method to retrieve the latest approved comments for a blog post.

Comment Repository

Open the `CommentRepository` class located at `src/Blogger/BlogBundle/Entity/Repository/CommentRepository.php` and replace its content with the following.

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

namespace Blogger\BlogBundle\Entity\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();
    }
}
```

The method we have created will retrieve comments for a blog post. To do this we need to add a where clause to our query. The where clause uses a named parameter that is set using the `setParameter()` method. You should always use parameters instead of setting the values directly in the query like so

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

In this example the value of `$blogId` will not be sanitized and could leave the query open to an **SQL injection** attack.

Blog Controller

Next we need to update the `show` action of the `Blog` controller to retrieve the comments for the blog. Update the `Blog`controller located at `src/Blogger/BlogBundle/Controller/BlogController.php` with the following.

```
// 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
    ));
}
```

We use the new method on the `CommentRepository` to retrieve the approved comments for the blog. The `$comments` collection is also passed into the template.

Blog show template

Now we have a list of comments for the blog we can update the blog show template to display the comments. We could simply place the rendering of the comments directly in the blog show template, but as comments are their own entity, it would be better to separate the rendering into another template, and include that template. This would allow us to reuse the comment rendering template elsewhere in the application. Update the blog show template located at`src/Blogger/BlogBundle/Resources/views/Blog/show.html.twig` with the following.

```
{# src/Blogger/BlogBundle/Resources/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 %}
```

You can see the use of a new Twig tag, the `include` tag. This will include the content of the template specified by`BloggerBlogBundle:Comment:index.html.twig`. We can also pass over any number of arguments to the template. In this case, we need to pass over a collection of `Comment` entities to render.

Comment show template

The `BloggerBlogBundle:Comment:index.html.twig` we are including above does not exist yet so we need to create it. As this is just a template, we don't need to create a route or a controller for this, we only need the template file. Create a new file located at `src/Blogger/BlogBundle/Resources/views/Comment/index.html.twig` and paste in the following.

```
{# src/Blogger/BlogBundle/Resources/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.created|date('l, F j, Y') }}</time></p>
        </header>
        <p>{{ comment.comment }}</p>
    </article>
{% else %}
    <p>There are no comments for this post. Be the first to comment...</p>
{% endfor %}
```

As you can see we iterate over a collection of `Comment` entities and display the comments. We also introduce one of the other nice Twig functions, the `cycle` function. This function will cycle through the values in the array you pass it as each iteration of the loop progresses. The current loop iteration value is obtained via the special `loop.index0` variable. This keeps a count of the loop iterations, starting at 0. There are a number of other **special variables** available when we are within a loop code block. You may also notice the setting of an HTML ID to the `article` element. This will allow us to later create permalinks to created comments.

Comment show CSS

Finally let's add some CSS to keep the comments looking stylish. Update the stylesheet located at`src/Blogger/BlogBundle/Resorces/public/css/blog.css` with the following.

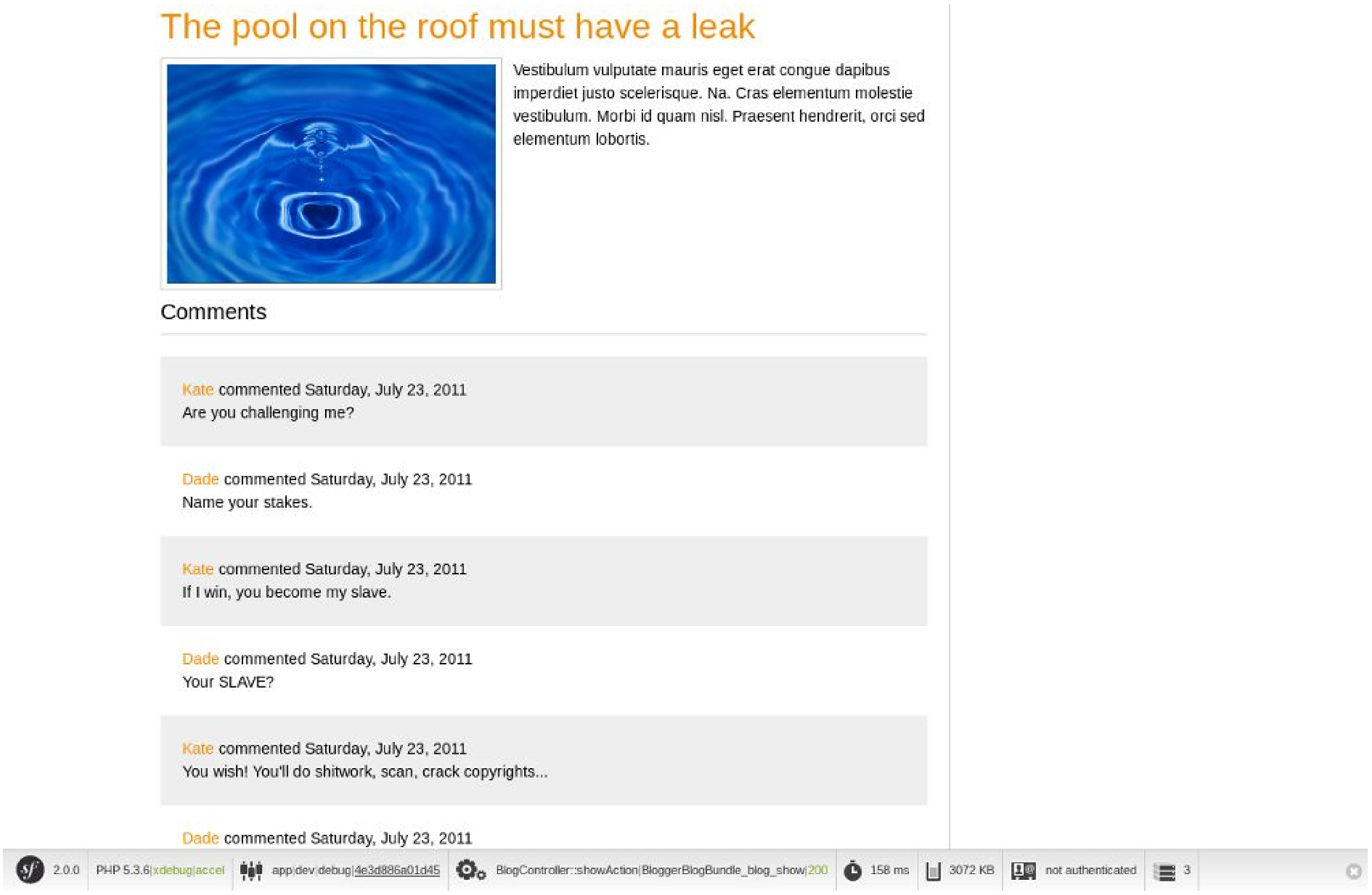
```
/** src/Blogger/BlogBundle/Resorces/public/css/blog.css **/
.comments { clear: both; }
.comments .odd { background: #eee; }
.comments .comment { padding: 20px; }
.comments .comment p { margin-bottom: 0; }
.comments h3 { background: #eee; padding: 10px; font-size: 20px; margin-bottom: 20px; clear: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

Note

If you are not using the symlink method for referencing bundle assets into the `web` folder you must re-run the assets install task now to copy over the changes to your CSS.

```
$ php app/console assets:install web
```

If you now have a look at one of the blog show pages, e.g. `http://symblog.dev/app_dev.php/2` you should see the blog comments output.



The pool on the roof must have a leak

Vestibulum vulputate mauris eget erat congue dapibus imperdiet justo scelerisque. Na. Cras elementum molestie vestibulum. Morbi id quam nisl. Praesent hendrerit, orci sed elementum lobortis.

Comments

Kate commented Saturday, July 23, 2011
Are you challenging me?

Dade commented Saturday, July 23, 2011
Name your stakes.

Kate commented Saturday, July 23, 2011
If I win, you become my slave.

Dade commented Saturday, July 23, 2011
Your SLAVE?

Kate commented Saturday, July 23, 2011
You wish! You'll do shitwork, scan, crack copyrights...

Dade commented Saturday, July 23, 2011

2.0.0
PHP 5.3.6|xdebug|accel
app|dev|debug|4e3d886a01d45
BlogController::showAction|BloggerBlogBundle_blog_show|200
158 ms
3072 KB
not authenticated
3

Adding Comments

The last part of this chapter will add the functionality for users to add comments to a blog post. This will be possible via a form on the blog show page. We have already been introduced to creating forms in Symfony2 when we created the contact form. Rather than creating the comment form manually, we can use Symfony2 to do this for us. Run the following task to generate the `CommentType` class for the `Comment` entity.

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

You'll notice again here, the use of the short hand version to specify the `Comment` entity.

Tip

You may have noticed the task `doctrine:generate:form` is also available. This is the same task just namespaced differently.

The generate form task has created the `CommentType` class located at `src/Blogger/BlogBundle/Form/CommentType.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';
    }
}
```

We have already explored what is happening here in the previous `EnquiryType` class. We could begin by customising this class now, but let's move onto displaying the form first.

Displaying the Comment Form

As we want the user to add comments from the show blog page, we could create the form in the `show` action of the `Blog`controller and render the form directly in the `show` template. However, it would be better to separate this code as we did with displaying the comments. The difference between showing the comments and displaying the comment form is the comment form needs processing, so this time a controller is required. This introduces a method slightly different to the above where we just included a template.

Routing

We need to create a new route to handle the processing of submitted forms. Add a new route to the routing file located at`src/Blogger/BlogBundle/Resources/config/routing.yml`.

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

The controller

Next, we need to create the new `Comment` controller we have referenced above. Create a file located at`src/Blogger/BlogBundle/Controller/CommentController.php` and paste in the following.

```
<?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;
    }

}
```

We create 2 actions in the `Comment` controller, one for `new` and one for `create`. The `new` action is concerned with displaying the comment form, the `create` action is concerned with processing the submission of the comment form. While this may seem like a big chuck of code, there is nothing new here, everything was covered in chapter 2 when we created the contact form. However, before moving on make sure you fully understand what is happening in the `Comment` controller.

Form Validation

We don't want users to be able to submit blog comments with blank `user` or `comment` values. To achieve this we look back to the validators we were introduced to in part 2 when creating the enquiry form. Update the `Comment` entity located at`src/Blogger/BlogBundle/Entity/Comment.php` with the following.

```
<?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'
        )));
    }

    // ..
}
```

The constraints ensure that both the user and comment members must not be blank. We have also set the `message` option for both constraints to override the default ones. Remember to add the namespace for `ClassMetadata` and `NotBlank` as shown above.

The view

Next we need to create the 2 templates for the `new` and `create` controller actions. First create a new file located at`src/Blogger/BlogBundle/Resources/views/Comment/form.html.twig` and paste in the following.

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

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

The purpose of this template is simple, it just renders the comment form. You'll also notice the `action` of the form is to `POST`to the new route we created `BloggerBlogBundle_comment_create`.

Next let's add the template for the `create` view. Create a new file located at`src/Blogger/BlogBundle/Resources/views/Comment/create.html.twig` and paste in the following.

```
{% 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 %}
```

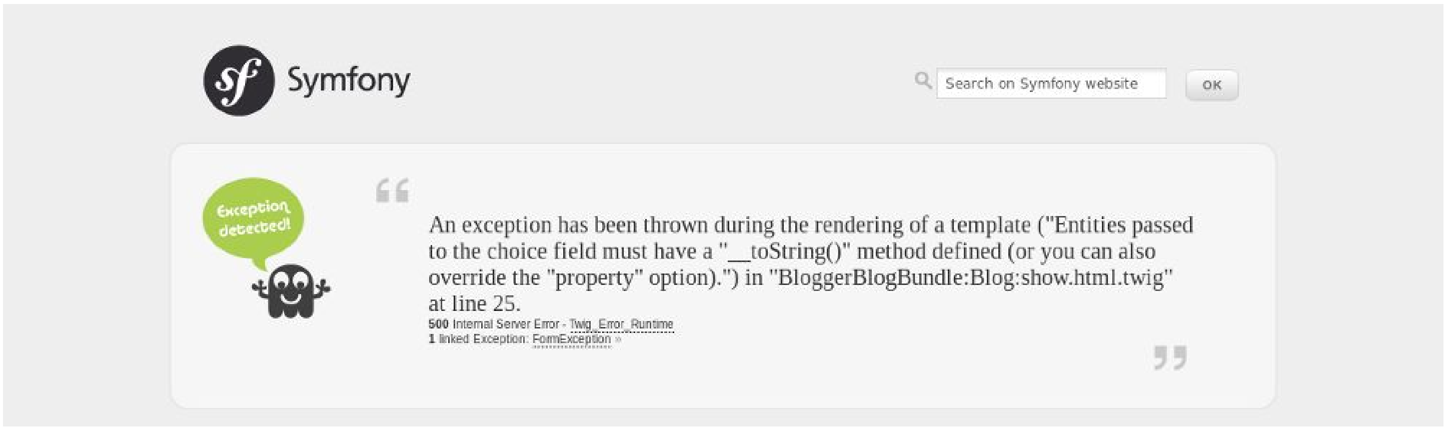
As the `create` action of the `Comment` controller deals with processing the form, it also needs to be able to display it, as there could be errors in the form. We reuse the `BloggerBlogBundle:Comment:form.html.twig` to render the actual form to prevent code duplication.

Now let's update the blog show template to render the add blog form. Update the template located at`src/Blogger/BlogBundle/Resources/views/Blog/show.html.twig` with the following.

```
{# src/Blogger/BlogBundle/Resources/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 %}
```

We use another new Twig tag here, the `render` tag. This tag will render the contents of a controller into the template. In our case we render the contents of the `BloggerBlogBundle:Comment:new` controller action.

If you now have a look at one of the blog show pages, such as `http://symblog.dev/app_dev.php/2` you'll notice a Symfony2 exception is thrown.



An exception has been thrown during the rendering of a template ("Entities passed to the choice field must have a "__toString()" method defined (or you can also override the "property" option).") in "BloggerBlogBundle:Blog:show.html.twig" at line 25.
500 Internal Server Error - Twig_Error_Runtime
1 linked Exception: FormException »

This exception is being thrown by the `BloggerBlogBundle:Blog:show.html.twig` template. If we look at line 25 of the`BloggerBlogBundle:Blog:show.html.twig` template we can see it's the following line showing that the problem actually exists in the process of embedding the `BloggerBlogBundle:Comment:create` controller.

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

If we look at the exception message further it gives us some more information about the nature of why the exception was caused.

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

This is telling us that a choice field that we are trying to render doesn't have a `__toString()` method set for the entity the choice field is associated with. A choice field is a form element that gives the user a number of choices, such as a `select`(drop down) element. You may be wondering where are we rendering a choice field in the comment form? If you look at the comment form template again you will notice we render the form using the `{{ form_widget(form) }}` Twig function. This function outputs the entire form in its basic form. So let's go back to the class the form is created from, the `CommentType`class. We can see that a number of fields are being added to the form via the `FormBuilder` object. In particular we are adding a `blog` field.

If you remember from chapter 2, we spoke about how the `FormBuilder` will try to guess the field type to output based on metadata related to the field. As we setup a relationship between `Comment` and `Blog` entities, the `FormBuilder` has guessed the comment should be a `choice` field, which would allow the user to specify the blog post to attach the comment to. That is why we have a `choice` field in the form, and why the Symfony2 exception is being thrown. We can fix this problem by implementing the `__toString()` method in the `Blog` entity.

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

Tip

The Symfony2 error messages are very informative when describing the problem that has occurred. Always read the error messages as they will usually make the debugging process a lot easier. The error messages also provide a full stack trace so you can see the steps that were taken to cause the error.

Now when you refresh the page you should see the comment form output. You will also notice that some undesirable fields have been output such as `approved`, `created`, `updated` and `blog`. This is because we did not customise the generated`CommentType` class earlier.

Tip

The fields being rendered all seem to be output as the correct type of fields. The `user` fields is an `text` field, the `comment` field is a `textarea`, the 2 `DateTime` fields are a number of `select` fields allowing us to specify the time, etc.

This is because of the ability of `FormBuilder` to guess the type of field the member it is rendering requires. It is able to do this based on the metadata you provide. As we have specified quite specific metadata for the`Comment` entity, the `FormBuilder` is able to make accurate guesses of the field types.

Let's now update this class located at `src/Blogger/BlogBundle/Form/CommentType.php` to output only the fields we need.

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

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

    // ..
}
```

Now when you refresh the page only the user and comment fields are output. If you were to submit the form now, the comment would not actually be saved to the database. That's because the form controller does nothing with the `Comment`entity if the form passes validation. So how do we persist the `Comment` entity to the database? You have already seen how to do this when creating `DataFixtures`. Update the `create` action of the `Comment` controller to persist the `Comment` entity to the database.

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

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

        if ($form->isValid()) {
            $em = $this->getDoctrine()
```