symblog

creating a blog in Symfony2

[Part 5] - Customising the view: Twig extensions, The sidebar and Assetic

Overview

This chapter will continue to build on the frontend of symblog. We will tweak the homepage to display information regarding comments for a blog post along with addressing SEO by adding the blog title to the URL. We will also begin work on the sidebar to add 2 common blog components; The Tag Cloud and the Latest Comments. We will explore the various environments in Symfony2 and learn how to run symblog in the production environment. The Twig templating engine will be extended to provide a new filter, and we introduce Assetic to manage the website asset files. At the end of this chapter you will have integrated comments into the homepage, have a Tag Cloud and Latest Comments component on the sidebar and will have used Assetic to manage the asset files. You will also have seen symblog running in the production environment.

The Homepage - Blogs and Comments

So far the homepage lists the latest blog entries but it doesn't give any information regarding the comments for those blogs. Now that we have the `Comment` entity built we can revisit the homepage to provide this information. As we have set up the link between `Blog` and `Comment` entities we know Doctrine 2 will be able to retrieve the comments for a blog (remember we added a `$comments` member to the `Blog` entity). Let's update the homepage view template located at`src/Blogger/BlogBundle/Resources/views/Page/index.html.twig` with the following.

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

{# .. #}

<footer class="meta">
    <p>Comments: {{ blog.comments|length }}</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>

{# .. #}
```

We have used the `comments` getter to retrieve the blog comments and then passed the collection through the Twig `length`filter. If you have a look at the homepage now via `http://symblog.dev/app_dev.php/` you will see the number of comments for each blog being displayed.

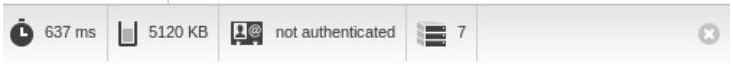
As explained above, we already informed Doctrine 2 that the `$comments` member of the `Blog` entity is mapped to the `Comment`entity. We achieved this in the previous chapter with the following metadata in the `Blog` entity located at`src/Blogger/BlogBundle/Entity/Blog.php`.

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

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

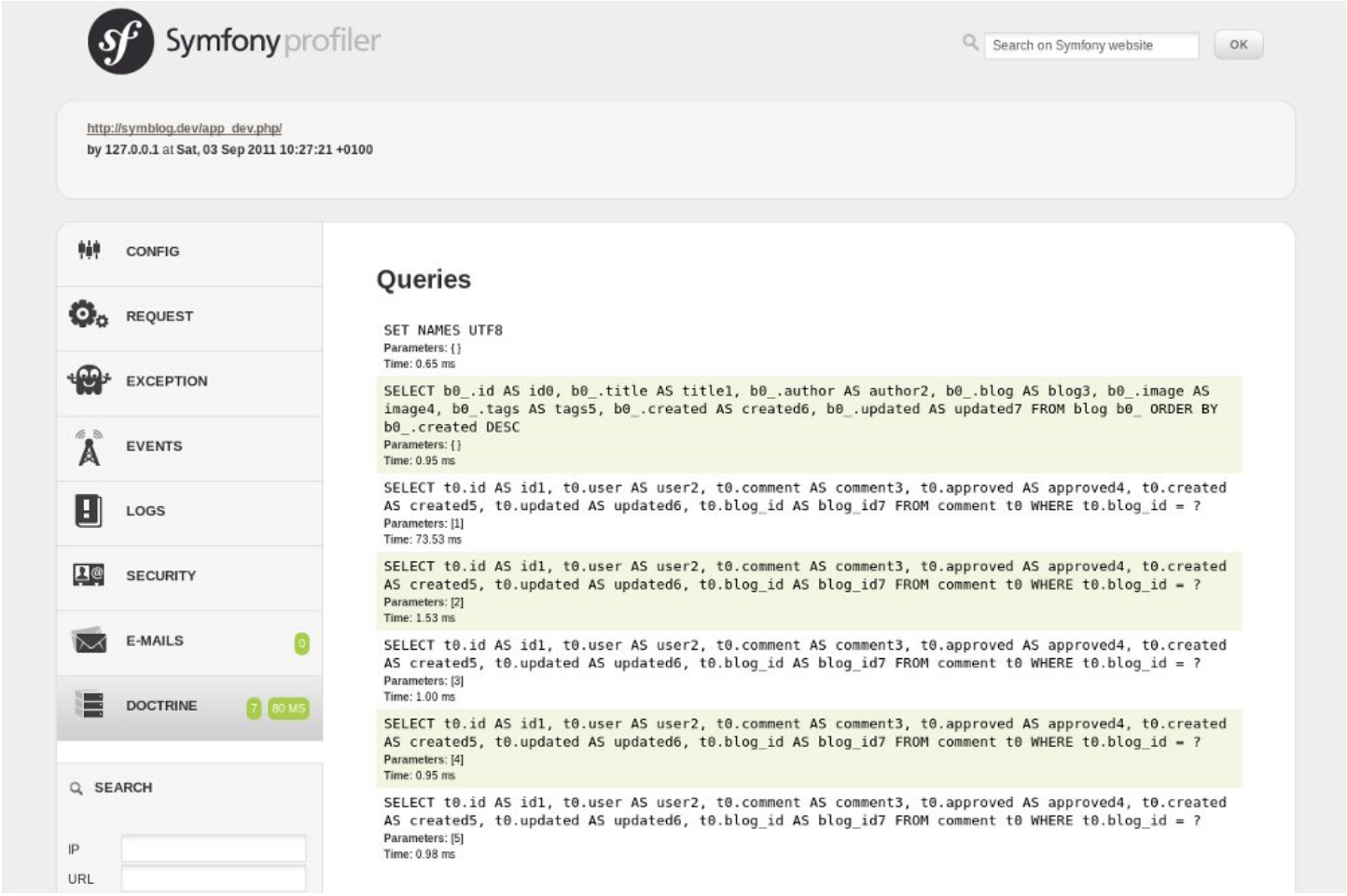
So we know Doctrine 2 is aware of the relationship between blogs and comments, but how did it populate the `$comments`member with the related `Comment` entities? If you remember back to the `BlogRepository` method we created (shown below) to get the homepage blogs we made no selection to retrieve the related `Comment` entities.

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

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();
}
```

However, Doctrine 2 uses a process called lazy loading where the `Comment` entities are retrieved from the database as and when required, in our case when the call to `{{ blog.comments|length }}` is made. We can demonstrate this process using the developer toolbar. We have already started to explore the basics of the developer toolbar and it's now time to introduce one of its most useful features, the Doctrine 2 profiler. The Doctrine 2 profiler can be accessed by clicking the last icon in the developer toolbar. The number next to this icon shows the number of queries executed on the database for the current HTTP request.



If you click the Doctrine 2 icon you will be presented with information regarding the queries that were executed by Doctrine 2 on the database for the current HTTP request.



As you can see in the above screen shot, there are a number of queries executed for a request to the homepage. The second query executed retrieves the blog entities from the database and is executed as a result of the `getLatestBlogs()`method on the `BlogRepository` class. Following this query you will notice a number of queries that pull comments from the database, one blog at a time. We can see this because of the `WHERE t0.blog_id = ?` in each of the queries, where the `?` is replaced by the Parameter value (the blog Id) on the following line. Each of these queries are as a result of the calls to `{{blog.comments }}` in the homepage template. Each time this function is executed, Doctrine 2 has to lazily load the `Comment`entities that relate to the `Blog` entity.

While lazy loading is very effective at retrieving related entities from the database, it's not always the most efficient way to perform this task. Doctrine 2 provides the ability to `join` related entities together when querying the database. This way we can pull the `Blog` and related `Comment` entities out from the database in one query. Update the `QueryBuilder` code in the`BlogRepository` located at `src/Blogger/BlogBundle/Repository/BlogRepository.php` to join on the comments.

```
// src/Blogger/BlogBundle/Repository/BLogRepository.php

public function getLatestBlogs($limit = null)
{
    $qb = $this->createQueryBuilder('b')
               ->select('b, c')
               ->leftJoin('b.comments', 'c')
               ->addOrderBy('b.created', 'DESC');

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

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

If you now refresh the homepage and examine the Doctrine 2 output in the developer toolbar you will notice the number of queries has dropped. You can also see the comment table has been joined to the blog table.

Lazy loading and joining related entities are both very powerful concepts but they need to be used correctly. The correct balance between the 2 needs to be found to ensure your application is running as efficiently as possible. At first it might seem great to join on every related entity so you never need to lazy load and your database query count will always remain low. However, it's important to remember that the more information you retrieve from the database, the more processing needs to be done by Doctrine 2 to hydrate this into the entity objects. More data also means more memory is used by the server to store the entity objects.

Before moving on let's make one minor addition to the homepage template for the number of comments we have just added. Update the homepage template located at `src/Blogger/BlogBundle/Resources/views/Page/index.html.twig` to add a link to show the blog comments.

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

{# .. #}

<footer class="meta">
    <p>Comments: <a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id }) }}#comments">{{ blog.comments|length }}</a></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>

{# .. #}
```

The Sidebar

Currently the sidebar of symblog is looking a bit empty. We will update this with 2 common blog components, the Tag Cloud and a list of the Latest Comments.

Tag Cloud

The Tag Cloud shows tags for each blog emphasized in a way that displays the more common tags bolder. To achieve this we need a way to retrieve all the tags for all the blogs.

Let's create some new methods in the `BlogRepository` class to do this. Update the `BlogRepository` class located at `src/Blogger/BlogBundle/Repository/BlogRepository.php` with the following.

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

public function getTags()
{
    $blogTags = $this->createQueryBuilder('b')
                     ->select('b.tags')
                     ->getQuery()
                     ->getResult();

    $tags = array();
    foreach ($blogTags as $blogTag)
    {
        $tags = array_merge(explode(",", $blogTag['tags']), $tags);
    }

    foreach ($tags as &$tag)
    {
        $tag = trim($tag);
    }

    return $tags;
}

public function getTagWeights($tags)
{
    $tagWeights = array();
    if (empty($tags))
        return $tagWeights;

    foreach ($tags as $tag)
    {
        $tagWeights[$tag] = (isset($tagWeights[$tag])) ? $tagWeights[$tag] + 1 : 1;
    }
    // Shuffle the tags
    uksort($tagWeights, function() {
        return rand() > rand();
    });

    $max = max($tagWeights);

    // Max of 5 weights
    $multiplier = ($max > 5) ? 5 / $max : 1;
    foreach ($tagWeights as &$tag)
    {
        $tag = ceil($tag * $multiplier);
    }

    return $tagWeights;
}
```

As the tags are stored in the database as comma separated values (CSV) we need a way to split them and return them as an array. This is achieved by the `getTags()` method. The `getTagWeights()` method is then able to use an array of tags to calculate the weight of each tag based on its popularity within the array. The tags are also shuffled to randomise their display on the page.

Now we are able to generate the Tag Cloud, we need to display it. Create a new action in the `PageController` located at`src/Blogger/BlogBundle/Controller/PageController.php` to handle the sidebar.

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

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

    $tags = $em->getRepository('BloggerBlogBundle:Blog')
               ->getTags();

    $tagWeights = $em->getRepository('BloggerBlogBundle:Blog')
                     ->getTagWeights($tags);

    return $this->render('BloggerBlogBundle:Page:sidebar.html.twig', array(
        'tags' => $tagWeights
    ));
}
```

The action is very simple, it uses the 2 new `BlogRepository` methods to generate the Tag Cloud, and passes this over to the view. Now let's create this view located at `src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig`.

```
{# src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig #}

<section class="section">
    <header>
        <h3>Tag Cloud</h3>
    </header>
    <p class="tags">
        {% for tag, weight in tags %}
            <span class="weight-{{ weight }}">{{ tag }}</span>
        {% else %}
            <p>There are no tags</p>
        {% endfor %}
    </p>
</section>
```

The template is also very simple. It just iterates over the various tags setting a class to the weight of the tag. The `for` loop introduces how to access the `key` and `value` pairs of the array, with `tag` being the key and `weight` being the value. There are a number of variations of how to use the `for` loop provided in the **Twig documentation**.

If you look back at the `BloggerBlogBundle` main layout template located at `src/Blogger/BlogBundle/Resources/views/layout.html.twig` you will notice we put a placeholder in for the sidebar block. Let's replace this now by rendering the new sidebar action. Remember from the previous chapter that the Twig `render` method will render the contents from a controller action, in this case the `sidebar` action of the `Page` controller.

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

{# .. #}

{% block sidebar %}
    {% render "BloggerBlogBundle:Page:sidebar" %}
{% endblock %}
```

Finally let's add the CSS for the tag cloud. Add a new stylesheet located at `src/Blogger/BlogBundle/Resources/public/css/sidebar.css`.

```
.sidebar .section { margin-bottom: 20px; }
.sidebar h3 { line-height: 1.2em; font-size: 20px; margin-bottom: 10px; font-weight: normal; background: #eee; padding: 5px;  }
.sidebar p { line-height: 1.5em; margin-bottom: 20px; }
.sidebar ul { list-style: none }
.sidebar ul li { line-height: 1.5em }
.sidebar .small { font-size: 12px; }
.sidebar .comment p { margin-bottom: 5px; }
.sidebar .comment { margin-bottom: 10px; padding-bottom: 10px; }
.sidebar .tags { font-weight: bold; }
.sidebar .tags span { color: #000; font-size: 12px; }
.sidebar .tags .weight-1 { font-size: 12px; }
.sidebar .tags .weight-2 { font-size: 15px; }
.sidebar .tags .weight-3 { font-size: 18px; }
.sidebar .tags .weight-4 { font-size: 21px; }
.sidebar .tags .weight-5 { font-size: 24px; }
```

As we have added a new stylesheet we need to include it. Update the `BloggerBlogBundle` main layout template located at`src/Blogger/BlogBundle/Resources/views/layout.html.twig` with the following.

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

{# .. #}

{% block stylesheets %}
    {{ parent() }}
    <link href="{{ asset('bundles/bloggerblog/css/blog.css') }}" type="text/css" rel="stylesheet" />
    <link href="{{ asset('bundles/bloggerblog/css/sidebar.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}

{# .. #}
```

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 new CSS file.

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

If you now refresh the symblog website you will see the Tag Cloud rendered in the sidebar. In order to get the tags to render with different weights, you may need to update the blog fixtures so some tags are used more than others.

Recent Comments

Now the Tag Cloud is in place, let's also add the Latest Comments component to the sidebar.

First we need a way to retrieve the latest comments for the blogs. To do this we will add a new method to the`CommentRepository` located at `src/Blogger/BlogBundle/Repository/CommentRepository.php`.

```
<?php
// src/BLogger/BLogBundle/Repository/CommentRepository.php

public function getLatestComments($limit = 10)
{
    $qb = $this->createQueryBuilder('c')
               ->select('c')
               ->addOrderBy('c.id', 'DESC');

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

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

Next update the sidebar action located at `src/Blogger/BlogBundle/Controller/PageController.php` to retrieve the latest comments and pass them over to the view.

```
// src/BLogger/BLogBundle/Controller/PageController.php

public function sidebarAction()
{
    // ..

    $commentLimit   = $this->container
                           ->getParameter('blogger_blog.comments.latest_comment_limit');
    $latestComments = $em->getRepository('BloggerBlogBundle:Comment')
                         ->getLatestComments($commentLimit);

    return $this->render('BloggerBlogBundle:Page:sidebar.html.twig', array(
        'latestComments'    => $latestComments,
        'tags'              => $tagWeights
    ));
}
```

You will notice we have used a new parameter called `blogger_blog.comments.latest_comment_limit` to limit the number of comments retrieved. To create this parameter update the config located at `src/Blogger/BlogBundle/Resources/config/config.yml` with the following.

```
# src/BLogger/BLogBundle/Resources/config/config.yml

parameters:
    # ..

    # BLogger max latest comments
    blogger_blog.comments.latest_comment_limit: 10
```

Finally we need to render the latest comments in the sidebar template. Update the template located at`src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig` with the following.

```
{# src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig #}

{# .. #}

<section class="section">
    <header>
        <h3>Latest Comments</h3>
    </header>
    {% for comment in latestComments %}
        <article class="comment">
            <header>
                <p class="small"><span class="highlight">{{ comment.user }}</span> commented on
                    <a href="{{ path('BloggerBlogBundle_blog_show', { 'id': comment.blog.id }) }}#comment-{{ comment.id }}">
                        {{ comment.blog.title }}
                    </a>
                    [<em><time datetime="{{ comment.created|date('c') }}">{{ comment.created|date('Y-m-d h:iA') }}</time></em>]
                </p>
            </header>
            <p>{{ comment.comment }}</p>
            </p>
        </article>
    {% else %}
        <p>There are no recent comments</p>
    {% endfor %}
</section>
```

If you now refresh the symblog website you will see the Latest Comments being displayed in the sidebar under the Tag Cloud.

Twig Extensions

So far we have been displaying the posted at dates for blog comments in a standard date format such as *2011-04-21*. A much nicer approach would be to display comment dates in terms of how long ago the comment was posted, such as*posted 3 hours ago*. We could add a method to the `Comment` entity to achieve this and change the templates to use this method instead of `{{ comment.created|date('Y-m-d h:iA') }}`.

As we may want to use this functionality elsewhere it would make more sense to move it out of the `Comment` entity. As transforming the date is specifically a view layer task, we should implement this using the Twig templating engine. Twig give us this ability by providing an Extension interface.

We can use the **extension** interface in Twig to extend the default functionality it provides. We are going to create a new Twig filter extension that can be used as follows.

```
{{ comment.created|created_ago }}
```

This would return the comment created date in a format such as *posted 2 days ago*.

The Extension

Create a file for the Twig extension located at `src/Blogger/BlogBundle/Twig/Extensions/BloggerBlogExtension.php` and updated with the following content.

```php
<?php
// src/Blogger/BlogBundle/Twig/Extensions/BloggerBlogExtension.php

namespace Blogger\BlogBundle\Twig\Extensions;

class BloggerBlogExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return array(
            'created_ago' => new \Twig_Filter_Method($this, 'createdAgo'),
        );
    }

    public function createdAgo(\DateTime $dateTime)
    {
        $delta = time() - $dateTime->getTimestamp();
        if ($delta < 0)
            throw new \InvalidArgumentException("createdAgo is unable to handle dates in the future");

        $duration = "";
        if ($delta < 60)
        {
            // Seconds
            $time = $delta;
            $duration = $time . " second" . (($time > 1) ? "s" : "") . " ago";
        }
        else if ($delta <= 3600)
        {
            // Mins
            $time = floor($delta / 60);
            $duration = $time . " minute" . (($time > 1) ? "s" : "") . " ago";
        }
        else if ($delta <= 86400)
        {
            // Hours
            $time = floor($delta / 3600);
            $duration = $time . " hour" . (($time > 1) ? "s" : "") . " ago";
        }
        else
        {
            // Days
            $time = floor($delta / 86400);
```

```
            $duration = $time . " day" . (($time > 1) ? "s" : "") . " ago";
        }

        return $duration;
    }

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

Creating the extension is quite simple. We override the `getFilters()` method to return any number of filters we want to be available. In this case we are creating the `created_ago` filter. This filter is then registered to use the `createdAgo` method, which simply transforms a `DateTime` object into a string representing the duration passed since the value stored in the `DateTime`object.

Registering the Extension

To make the Twig extension available we need to update the services file located at`src/Blogger/BlogBundle/Resources/config/services.yml` with the following.

```
services:
    blogger_blog.twig.extension:
        class: Blogger\BlogBundle\Twig\Extensions\BloggerBlogExtension
        tags:
            - { name: twig.extension }
```

You can see this is registering a new service using the `BloggerBlogExtension` Twig extension class we have just created.

Updating the view

The new Twig filter is now ready to be used. Let's update the sidebar Latest Comments list to use the `created_ago` filter. Update the sidebar template located at `src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig` with the following.

```
{# src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig #}

{# .. #}

<section class="section">
    <header>
        <h3>Latest Comments</h3>
    </header>
    {% for comment in latestComments %}
        {# .. #}
        <em><time datetime="{{ comment.created|date('c') }}">{{ comment.created|created_ago }}</time></em>
        {# .. #}
    {% endfor %}
</section>
```

If you now point your browser to `http://symblog.dev/app_dev.php/` you will see the latest comment dates are using the Twig filter to render the duration since the comment was posted.

Let's also update the comments listed on the blog show page to use the new filter. Replace the content in the template located at `src/Blogger/BlogBundle/Resources/views/Comment/index.html.twig` with 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|created_ago }}</time></p>
        </header>
        <p>{{ comment.comment }}</p>
    </article>
{% else %}
    <p>There are no comments for this post. Be the first to comment...</p>
{% endfor %}
```

> **Tip**
>
> There are a number of useful Twig extensions available via the **Twig-Extensions** library on GitHub. If you create a useful extension send over a pull request for this repository and it may get included for other people to use.

Slugifying the URL

Currently the URL for each blog post only shows the blog ID. While this is perfectly acceptable from a functional point of view, it's not great for SEO. For example, the URL `http://symblog.dev/1` doesn't give any information away about the content of the blog, something like `http://symblog.dev/1/a-day-with-symfony2` would be much better. To achieve this we will slugify the blog title and use it as part of this URL. Slugifying the title will remove all non ASCII characters and replace them with a `-`.

Update the routing

To begin let's modify the routing rule for the blog show page to add the slug component. Update the routing rule located at `src/Blogger/BlogBundle/Resources/config/routing.yml`

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

BloggerBlogBundle_blog_show:
    pattern:  /{id}/{slug}
    defaults: { _controller: BloggerBlogBundle:Blog:show }
    requirements:
        _method:  GET
        id: \d+
```

The controller

As with the existing `id` component, the new `slug` component will be passed into the controller action as an argument, so let's update the controller located at `src/Blogger/BlogBundle/Controller/BlogController.php` to reflect this.

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

public function showAction($id, $slug)
{
    // ..
}
```

> **Tip**
>
> The order in which the arguments are passed into the controller action doesn't matter, only the name of them does. Symfony2 is able to match up the routing arguments with the parameter list for us. While we haven't yet used default component values it's worth mentioning them here. If we added another component onto the routing rule we can specify a default value for the component using the `defaults` option.

```
BloggerBlogBundle_blog_show:
    pattern:  /{id}/{slug}/{comments}
    defaults: { _controller: BloggerBlogBundle:Blog:show, comments: true }
    requirements:
        _method:  GET
        id: \d+
```

```
public function showAction($id, $slug, $comments)
{
    // ..
}
```

Using this method, a request to `http://symblog.dev/1/symfony2-blog` would result in `$comments` being set to true in the `showAction`.

Slugifying the title

As we want to generate the slug from the blog title, we will auto generate the slug value. We could simply perform this operation at run time on the title field but instead we will store the slug in the `Blog` entity and persist it to the database.

Updating the Blog entity

Let's add a new member to the `Blog` entity to store the slug. Update the `Blog` entity located at `src/Blogger/BlogBundle/Entity/Blog.php`

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

class Blog
{
    // ..

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

    // ..
}
```

Now generate the accessors for the new `$slug` member. As before run the following task.

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

Next, let's update the database schema.

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

To generate the slug value we will use the slugify method from the symfony 1 **Jobeet** tutorial. Add the `slugify` method to the the `Blog` entity located at `src/Blogger/BlogBundle/Entity/Blog.php`

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

public function slugify($text)
{
    // replace non letter or digits by -
    $text = preg_replace('#[^\\pL\d]+#u', '-', $text);

    // trim
    $text = trim($text, '-');

    // transliterate
    if (function_exists('iconv'))
    {
        $text = iconv('utf-8', 'us-ascii//TRANSLIT', $text);
    }

    // lowercase
    $text = strtolower($text);

    // remove unwanted characters
    $text = preg_replace('#[^-\w]+#', '', $text);

    if (empty($text))
    {
        return 'n-a';
    }

    return $text;
}
```

As we want to auto generate the slug from the title we can generate the slug when the value of the title is set. For this we can update the `setTitle` accessor to also set the value of the slug. Update the `Blog` entity located at `src/Blogger/BlogBundle/Entity/Blog.php` with the following.

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

public function setTitle($title)
{
    $this->title = $title;

    $this->setSlug($this->title);
}
```

Next update the `setSlug` method to slugify the slug before it is set.

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

public function setSlug($slug)
{
    $this->slug = $this->slugify($slug);
}
```

Now reload the data fixtures to generate the blog slugs.

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

Updating the generated routes

Finally we need to update the existing calls for generating routes to the blog show page. There are a number of locations where this needs to be updated.

Open the homepage template located at `src/Blogger/BlogBundle/Resources/views/Page/index.html.twig` and replace its contents with the following. There have been 3 edits to the generation of the `BloggerBlogBundle_blog_show` route in this template. The edits simply pass in the blog slug to the Twig `path` function.

```
{# 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, 'slug': blog.slug }) }}">{{ 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, 'slug': blog.slug }) }}">Continue reading...</a></p>
            </div>

            <footer class="meta">
                <p>Comments: <a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id, 'slug': blog.slug }) }}#comments">{{ blog.comments|length }}</a></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 %}
```

Also, one update needs to be made to the Latest Comments section of the sidebar template located at`src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig`.

```
{# src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig #}

{# .. #}

<a href="{{ path('BloggerBlogBundle_blog_show', { 'id': comment.blog.id, 'slug': comment.blog.slug }) }}#comment-{{ comment.id }}">
    {{ comment.blog.title }}
</a>

{# .. #}
```

Finally the `createAction` of the `CommentController` needs to be updated when redirecting to the blog show page on a successful comment posting. Update the `CommentController` located at `src/Blogger/BlogBundle/Controller/CommentController.php` with the following.

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

public function createAction($blog_id)
{
    // ..

    if ($form->isValid()) {
        // ..

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

    // ..
}
```

Now if you navigate to the symblog homepage at `http://symblog.dev/app_dev.php/` and click one of the blog titles you will see the blog slug has been appended to the end of the URL.

Environments

Environments are a very powerful, yet simple feature provided by Symfony2. You may not be aware, but you have been using environments from part 1 of this tutorial. With environments we can configure various aspects of Symfony2 and the application to run differently depending on the specific needs during the applications life cycle. By default Symfony2 comes configured with 3 environments:

1. `dev` - Development
2. `test` - Test
3. `prod` - Production

The purpose of these environments is self explanatory, but what about these environments would be configured differently for their individual needs. When developing the application it's useful to have the developer toolbar on screen with descriptive exceptions and errors being displayed, while in production you don't want any of this. In fact, having this information displayed would be a security risk as a lot of details regarding the internals of the application and the server would be exposed. In production it would be better to display customised error pages with simplified messages, while quietly logging this information to text files. It would also be useful to have the caching layer enabled to ensure the application is running at its best. Having the caching layer enabled in the `development` environment would be a pain as you would need to empty the cache each time you made changes to config files, etc.

The other environment is the `test` environment. This is used when running tests on the application such as unit or functional test. We haven't covered testing yet, but rest assured it will be covered in depth in the coming chapters.

Front Controllers

So far through this tutorial we have been using the `development` environment only. We have been specifying to run in the`development` environment by using the `app_dev.php` front controller when making requests to symblog, eg`http://symblog.dev/app_dev.php/about`. If we have a look at the front controller located at `web/app_dev.php` you will see the following line:

```
$kernel = new AppKernel('dev', true);
```

This line is what kick starts Symfony2 going. It instantiates a new instance of the Symfony2 `AppKernel` and sets the environment to `dev`.

In contrast, if we look at the front controller for the `production` environment located at `web/app.php` we see the following:

```
$kernel = new AppKernel('prod', false);
```

You can see the `prod` environment is passed into the `AppKernel` in this instance.

The test environment is not supposed to be run via the web browser which is why there is no `app_test.php` front controller.

Configuration Settings

We have seen above how the front controllers are utilised to change the environment the application runs under. Now we will explore how the various settings are modified while running under each environment. If you have a look at the files in`app/config` you will see a number of `config.yml` files. Specifically there is one main one, called `config.yml` and 3 others all suffixed with the name of an environment; `config_dev.yml`, `config_test.yml` and `config_prod.yml`. Each of these files is loaded depending on the current environment. If we explore the `config_dev.yml` file you will see the following lines at the top.

```
imports:
    - { resource: config.yml }
```

The `imports` directive will cause the `config.yml` file to be included into this file. The same `imports` directive can be found at the top of the other 2 environment config files, `config_test.yml` and `config_prod.yml`. By including a common set of config settings defined in `config.yml` we are able to override specific settings for each environment. We can see in the `development` config file located at `app/config/config_dev.yml` the following lines configuring the use of the developer toolbar.

```
# app/config/config_dev.yml

web_profiler:
    toolbar: true
```

This setting is absent from the `production` config file as we don't want the developer toolbar displayed.

Running in Production

For those of you eager to see your site running in the `production` environment now is the time.

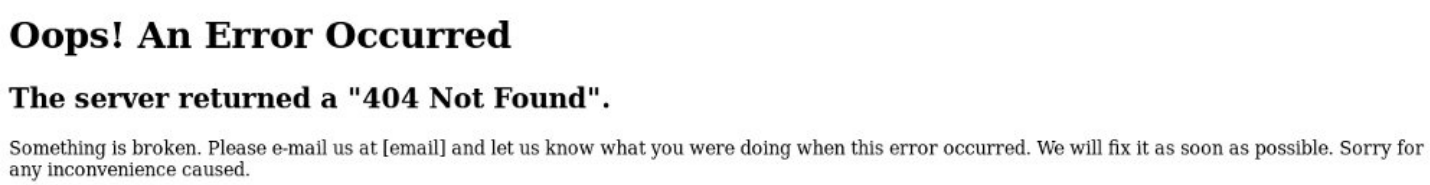
First we need to clear the cache using one of the Symfony2 tasks.

```
$ php app/console cache:clear --env=prod
```

Now point your browser to `http://symblog.dev/`. Notice the `app_dev.php` front controller is missing.

> **Note**
>
> For those of you using the Dynamic Virtual Hosts configuration as linked to in part 1, you will need to add the following to the .htaccess file located at `web/.htaccess`.
>
> ```
> <IfModule mod_rewrite.c>
>     RewriteBase /
>     # ...
> </IfModule>
> ```

You will notice the site looks pretty much the same, but a few important features are different. The developer toolbar is now gone and the detailed exception message are no longer displayed, try going to `http://symblog.dev/999`.



The detailed exception message has been replaced by a simplified message informing the user of the problem. These exception screens can be customised to match the look and feel of your application. We will explore this in later chapters.

Furthermore, you'll notice the `app/logs/prod.log` file is filling up with logs regarding the execution of the application. This is a useful point of call when you have issues with the application in `production` as errors and exceptions won't be displayed on the screen any more.

> **Tip**
>
> How did the request to `http://symblog.dev/` end up being routed through the file `app.php`? I'm sure you're all used to creating files such as `index.html` and `index.php` that act as the site's index, but how would `app.php` become this? This is thanks to a RewriteRule in the file `web/.htaccess`
>
> ```
> RewriteRule ^(.*)$ app.php [QSA,L]
> ```
>
> We can see that this line has a regular expression that matches any text, denoted by `^(.*)$` and passes this to `app.php`.
>
> You may be on an Apache server that doesn't have the `mod_rewrite.c` enabled. If this is the case you can simply add `app.php` to the URL such as `http://symblog.dev/app.php/`.

While we have covered the basics of the `production` environment, we have not covered many other `production` related tasks such as customising error pages, and deployment to the `production` server using tools such as **capifony**. These topics will be covered in later chapters.

Creating New Environments

Finally it's worth noting that you can setup your own environments easily in Symfony2. For example, you may want a staging environment that would run on the production server, but output some of the debugging information such as exceptions. This would allow the platform to be tested manually on the actual production server as production and development configurations of servers can differ.

While creating a new environment is a simple task, it is outside the scope of this tutorial. There is an excellent **article** in the Symfony2 cookbook that covers this.

Assetic

The Symfony2 Standard Distribution is bundled with a library for assets management called **Assetic**. The library was developed by **Kris Wallsmith** and was inspired by the Python library **webassets**.

Assetic deals with 2 parts of asset management, the assets such as images, stylesheets and JavaScript and the filters that can be applied to these assets. These filters are able to perform useful tasks such as minifying your CSS and JavaScript, passing **CoffeeScript** files through the CoffeeScript compiler, and combining asset files together to reduce the number of HTTP requests made to the server.

Currently we have been using the Twig `asset` function to include assets into the template as follows.

```
<link href="{{ asset('bundles/bloggerblog/css/blog.css') }}" type="text/css" rel="stylesheet" />
```

The calls to the `asset` function will be replaced by Assetic.

Assets

The Assetic library describes an asset as follows:

An Assetic asset is something with filterable content that can be loaded and dumped. An asset also includes metadata, some of which can be manipulated and some of which is immutable.

Put simply, the assets are the resources the application uses such as stylesheets and images.

To enable Assetic for the `BloggerBlogBundle` we have to change the `app/config/config.yml` as follows.

```
# ..
assetic:
    bundles:    [BloggerBlogBundle]
# ..
```

This will enable Assetic just for the `BloggerBlogBundle` and will require adjustments whenever a new bundle needs to use Assetic. We can however completely remove the `bundles` line and enable it for all future bundles aswell.

Stylesheets

Let's begin by replacing the current calls to `asset` for the stylesheets in the `BloggerBlogBundle` main layout template. Update the content of the template located at `src/Blogger/BlogBundle/Resources/views/layout.html.twig` with the following.

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

{# .. #}

{% block stylesheets %}
    {{ parent () }}

    {% stylesheets
        '@BloggerBlogBundle/Resources/public/css/*'
    %}
        <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
    {% endstylesheets %}
{% endblock %}

{# .. #}
```

We have replaced the 2 previous links for CSS files with some Assetic functionality. Using `stylesheets` from Assetic we have specified that all CSS files in the location `src/Blogger/BlogBundle/Resources/public/css` should be combined into 1 file and then output. Combining files is a very simple but effective way to optimise your website frontend by reducing the number of files needed. Fewer files means fewer HTTP requests to the server. While we used the `*` to specify all files in the `css`directory we could have simply listed each file individually as follows.

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

{# .. #}

{% block stylesheets %}
    {{ parent () }}

    {% stylesheets
        '@BloggerBlogBundle/Resources/public/css/blog.css'
        '@BloggerBlogBundle/Resources/public/css/sidebar.css'
    %}
        <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
    {% endstylesheets %}
{% endblock %}

{# .. #}
```

The end result in both cases is the same. The first option using the `*` ensures that when new CSS files are added to the directory, they will always be included in the combined CSS file by Assetic. This may not be the desired functionality for your website, so use either method above to suit your needs.

If you have a look at the HTML output via `http://symblog.dev/app_dev.php/` you will see the CSS has been included something like this (Notice we are running back in the `development` environment again).

```
<link href="/app_dev.php/css/d8f44a4_part_1_blog_1.css" rel="stylesheet" media="screen" />
<link href="/app_dev.php/css/d8f44a4_part_1_sidebar_2.css" rel="stylesheet" media="screen" />
```

Firstly you may be wondering why there are 2 files. Above it was stated that Assetic would combine the files into 1 CSS file. This is because we are running symblog in the `development` environment. We can ask Assetic to run in non-debug mode by setting the debug flag to false as follows.

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

{# .. #}

{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/*'
    debug=false
%}
    <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
{% endstylesheets %}

{# .. #}
```

Now if you look at the rendered HTML you will see something like this.

```
<link href="/app_dev.php/css/3c7da45.css" rel="stylesheet" media="screen" />
```

If you view the contents of this file you will see the 2 CSS files, `blog.css` and `sidebar.css` have been combined into 1 file. The filename given to the generated CSS file is randomly generated by Assetic. If you would like to control the name given to the generated file use the `output` option as follows.

```
{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/*'
    output='css/blogger.css'
%}
    <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
{% endstylesheets %}
```

Before you continue remove the debug flag from the previous snippet as we want to resume default behavior on the assets.

We also need to update the applications base template located at `app/Resources/views/base.html.twig`.

```
{# app/Resources/views/base.html.twig #}

{# .. #}

{% block stylesheets %}
    <link href='http://fonts.googleapis.com/css?family=Irish+Grover' rel='stylesheet' type='text/css'>
    <link href='http://fonts.googleapis.com/css?family=La+Belle+Aurore' rel='stylesheet' type='text/css'>
    {% stylesheets
        'css/*'
    %}
        <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
    {% endstylesheets %}
{% endblock %}

{# .. #}
```

JavaScripts

While we currently don't have any JavaScipt files in our application, its usage in Assetic is much the same as using stylesheets.

```
{% javascripts
    '@BloggerBlogBundle/Resources/public/js/*'
%}
    <script type="text/javascript" src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Filters

The real power in Assetic comes from the filters. Filters can be applied to assets or collections of assets. There are a large number of filters provided within the core of the library including the following common filters:

1. `CssMinFilter`: minifies CSS
2. `JpegoptimFilter`: optimize your JPEGs
3. `Yui\CssCompressorFilter`: compresses CSS using the YUI compressor
4. `Yui\JsCompressorFilter`: compresses JavaScript using the YUI compressor
5. `CoffeeScriptFilter`: compiles CoffeeScript into JavaScript

There is a full list of available filters in the **Assetic Readme**.

Many of these filters pass the actual task onto another program or library, such as YUI Compressor, so you may need to install/configure the appropriate libraries to use some of the filters.

Download the **YUI Compressor**, extract the archive and copy the file located in the `build` directory to`app/Resources/java/yuicompressor-2.4.6.jar`. This assumes you downloaded the `2.4.6` version of the YUI Compressor. If not change your version number accordingly.

Next we will configure an Assetic filter to minify the CSS using the YUI Compressor. Update the application config located at `app/config/config.yml` with the following.

```
# app/config/config.yml

# ..

assetic:
    filters:
        yui_css:
            jar: %kernel.root_dir%/Resources/java/yuicompressor-2.4.6.jar

# ..
```

We have configured a filter called `yui_css` that will use the YUI Compressor Java executable we placed in the applications resources directory. In order to use the filter you need to specify which assets you want the filter applied to. Update the template located at `src/Blogger/BlogBundle/Resources/views/layout.html.twig` to apply the `yui_css` filter.

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

{# .. #}

{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/*'
    output='css/blogger.css'
    filter='yui_css'
%}
    <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
{% endstylesheets %}

{# .. #}
```

Now if you refresh the symblog website and view the files output by Assetic you will notice they have been minified. While minification is great for production servers, it can make debugging difficult, especially when JavaScript is minified. We can disable the minification when running in the `development` environment by prefixing the filter with a `?` as follows.

```
{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/*'
    output='css/blogger.css'
    filter='?yui_css'
%}
    <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
{% endstylesheets %}
```

Dumping the assets for production

In production we can dump the asset files using Assetic so they become actual resources on disk ready to be served by the web server. The process of creating the assets through Assetic with every page request can be quite slow, especially when filters are being applied to the assets. Dumping the assets for `production` ensures that Assetic is not used to serve the assets and instead the pre-processed asset files are served directly by the web server. Run the following task to create dump the asset files.

```
$ app/console --env=prod assetic:dump
```

You will notice a number of CSS files were created at `web/css` including the combined `blogger.css` file. Now if you run the symblog website in the `production` environment via `http://symblog.dev/` the files will be served directly from this folder.

> **Note**
>
> If you dump the asset files to disk and want to revert back to the `development` environment, you will need to clean up the created asset files in `web/` to allow Assetic to recreate them.

Additional Reading

We have only scratched the surface at what Assetic can perform. There are more resources available online especially in the Symfony2 cookbook including:

How to Use Assetic for Asset Management

How to Minify JavaScripts and Stylesheets with YUI Compressor

How to Use Assetic For Image Optimization with Twig Functions

How to Apply an Assetic Filter to a Specific File Extension

There are also a number of great article written by **Richard Miller** including:

Symfony2: Using CoffeeScript with Assetic

Symfony2: A Few Assetic Notes

Symfony2: Assetic Twig Functions

> **Tip**
>
> It's worth mentioning here that Richard Miller has a collection of excellent articles regarding a number of Symfony2 areas on his site including Dependency Injection, Services and the above mentioned Assetic guides. Just search for posts tagged with **symfony2**

Conclusion

We have covered a number of new areas with regards to Symfony2 including the Symfony2 environments and how to use the Assetic asset library. We also made improvements to the homepage and added some components to the sidebar.

In the next chapter we will move on to testing. We will explore both unit and functional testing using PHPUnit. We will see how Symfony2 comes complete with a number of classes to assist in writing functional tests that simulate web requests, allow us to populate forms and click links and then inspect the returned response.