Symfony Best Practices

With <3 from SymfonyCasts

Chapter 1: Symfony: Keep it Simple with @Route and Templates

SYMFONY: KEEP IT SIMPLE WITH @ROUTE AND TEMPLATES¶

Hey Symfony world. So you probably saw that Symfony released these best practices and I'll admit I was partially responsible for these. But apart from that, I'm actually really really excited about them because they're going to allow us reduce the complexity that we have in our application.

Note

The best practices are aimed at *your* non-shared application code. If you *are* building something that you need to share internally or to the world, you'll want to do a little more work. See Best Practices for Reusable Bundles

Complexity Versus Simplicity Versus Easy¶

Last week at Symfony Live, I spent my entire presentation actually talking about complexity and how we can reduce it in our applications. The opposite of complexity is simplicity which means "easily understood". Simplicity is already something that we already really want in our own code. We want to come back to our code in 6 months and say "this make sense to me" - not "what was I thinking here? I don't remember, I need to dive in and figure out what I was thinking".

I wanted to show a few of my favorite best practices that are going to reduce complexity. In particular a few things that are going to give us all less directories, less files and will make our projects a lot smaller and easier to navigate.

Our Project¶

I'm starting with a fresh Symfony 2.5 project because all of the changes we're going to talk about are things that can be done in any version of Symfony. The project has an `AppBundle` in it. Now notice the interesting thing here is that we don't have a vendor namespace:

```
src/
    AppBundle/
        AppBundle.php
        DataFixtures/
        Entity/
```

That's normally the `src/MyCompany/AppBundle`. This just isn't necessary in your own projects. You're not going to collide with anybody else's namespaces because all reusable third-party bundles *do* have a vendor namespace. So don't put one in yours, it just makes things longer.

Other than that, we have a really simple `Post` entity:

```
// src/AppBundle/Entity/Post.php
// ...

/**
 * @ORM\Entity
 */
class Post
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

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

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

    // ...
}
```

I've added some data fixtures using Alice and Faker. I'll talk about that another time, but that's a really great way to do your fixtures. And the only other changes is in `app/config/routing.yml`. I already have a line that import annotations from my `Controller/` directory"

```
# app/config/routing.yml
app_bundle_annotations:
    resource: "@AppBundle/Controller"
    type: annotation
```

This is likely a change that you'll see out of the box in Symfony 2.6.

I don't have that directory yet, so let's go ahead and create it otherwise Symfony will throw an error at us:

> mkdir src/AppBundle/Controller

I've already initialize the database and loaded my fixtures. I have my build-in web server already running so let's go and try it out:

```
composer install
php app/console doctrine:database:create
php app/console doctrine:schema:create
php app/console doctrine:fixtures:load
php app/console server:run
```

```
http://localhost:8000
```

And there is our beautiful 404 page, because of course we don't have a homepage yet.

Creating the Simplest Page Ever¶

So let's go ahead and create a page. The first page I want to create is something that lists all posts. I'm using PHPStorm with the awesome Symfony2 plugin so I have that nice Symfony2 controller option there. But if you don't, just create the controller by hand:

```php
// src/AppBundle/Controller/PostController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class PostController extends Controller
{
}
```

The `@Route` Annotation¶

So instead of having an extra routing.yml file, we're just going to use the `@Route` annotation and give it a path. Of course we need to remove the `$name` argument since we don't have that in our route anymore and I'll give it an inspirational die statement so we can make sure things are working:

```php
// src/AppBundle/Controller/PostController.php
// ...

/**
 * @Route("/posts")
 */
public function indexAction()
{
    die('it works!');
}
```

Now, as many of you know, every time you have an annotation, you need to have a `use` statement for it. So I'll let PHPStorm help me here and auto-complete that `use` statement. But you can also just go Google for SensioFrameworkExtraBundle, which is what gives us the `@Route` annotation. Scroll down a little bit and you'll see all of the `use` statements you'll need if you use this library:

```php
// src/AppBundle/Controller/PostController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class PostController extends Controller
{
    // indexAction lives here ...
}
```

So we have the `@Route` annotation, we have our control method, so lets try going to `/post` and it works!

```
http://localhost:8000/posts
```

So no surprises there: the `@Route` annotation is all we need.

Simple Template Organization¶

So let's finish this page. It should be fairly straightforward: we're going to use Doctrine to query for all the posts and then pass them into a template:

```php
/**
 * @Route("/posts")
 */
public function indexAction()
{
    $posts = $this->getDoctrine()
        ->getRepository('AppBundle:Post')
        ->findAll();

    return $this->render('Post/index.html.twig', array(
        'posts' => $posts,
    ));
}
```

Now, notice that my template name does *not* have any colons in it. Normally we have this `AppBundle:Post:index.html.twig` thing. One of my favorite new best practices is to store your templates in the `app/Resources/views` directory. And when you do this, you don't need any colons: you can just say `Post/index.html.twig` and it's going to look for that in the `app/Resources/views` directory.

Note

I'm using an upper-case `Post`, but the best-practices say to use a lowercase `post`. I actually like the lowercase better, but choose whatever you want.

I'll create a template and of course make it extend `base.html.twig`. And for the same reason here we don't need the `::` before. We can just say `base.html.twig` and it's going to look in the `app/Resources/views` directory:

```
{# app/Resources/views/Post/index.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
<h1>POSTS!</h1>

<ul>
   {% for post in posts %}
      <li>
         {{ post.title }}
      </li>
   {% endfor %}
</ul>
{% endblock %}
```

Now this may seem like a small detail, but there's 2 massive advantages to this. One, nobody liked or understood the colon syntax, especially beginners. I hated teaching it: every time I told them the `AppBundle:Post:index.html.twig` thing, it didn't make sense to anybody. The second thing is that we no longer have our templates spread out across our project or our bundles. So if you have a frontend developer working with you, they can easily find those templates because they're all sitting in one directory. A lot of times complexity is about perceived complexity: the more files and directories you have, the harder things are going to luck.

And no surprises, when we refresh, we have a working page. So one thing I want to highlight is that we only touched two files: our controller and our template.

Creating the Show Page¶

So now I want to create a page that is going to show just one blog post, and it's going to be even easier. Just like before, we'll start with `@Route`. The only difference here is that we'll have the `{id}` wildcard. And as you already know we're going to map that to an `$id` argument in our controller. And because I love `die` statements, we'll try that just to test it out:

```
// src/AppBundle/Controller/PostController.php
// ...

/**
 * @Route("/posts/{id}")
 */
public function showAction($id)
{
   die('Mr Testers');
}
```

Add an id on the end of the URL and there's our `die` statement:

```
http://localhost:8000/posts/5
```

The (In)Famous ParamConverter Trick¶

So now I want to show you one controversial trick. Normally if we have `{id}` in the URL, then we have a `$id` argument. But you can also *change* that argument if you type-hint it with `Post`, which is our entity. Then Doctrine is going to automatically query for that `Post` based on the `{id}` in the URL. And if it doesn't find one, it's going to throw a 404 page:

```
// src/AppBundle/Controller/PostController.php
// ...

/**
 * @Route("/posts/{id}")
 */
public function showAction(Post $post)
{
    var_dump($post);die;
}
```

And in this case, you can see it works perfectly. This comes from the ParamConverter of the SensioFrameworkExtraBundle and the only gotcha is that the name of your wildcard - so `{id}` for us - needs to match up with the property. So we have an `{id}` wildcard and we have an `id` property. If we change that to be `{postId}`, it's not going to work because it doesn't match our property name. Yes there *are* ways to configure the `ParamConverter` to figure this all out. But right now the configuration is actually really ugly, so I use this when it's easy and if it's not easy I just query myself. It's not a big deal.

Let's finish this up. We'll render a template. Notice the controller is basically only one line, which is nice. And then we'll create a template just to make sure that things are actually working. Print out the title, print out the contents and refresh to see some nice Latin on the screen:

```
{# app/Resources/views/Post/show.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
<h1>{{ post.title }}</h1>

<div>
    {{ post.contents }}
</div>
{% endblock %}
```

Route Names and Requirements¶

The other common thing that routes need are names. And actually right now, our routes *do* have a name. If we go over to `router:debug`, we're going to see that Symfony has given an auto-generated names to each of our routes, which is fine, but I don't exactly trust that:

```
php app/console router:debug
```

So the minute I actually need to link to one of these pages, I'm going to pass a `name` option to the `@Route` annotation to give it a specific name:

```
/**
 * @Route("/posts/{id}", name="post_show")
 */
public function showAction(Post $post)
{
    // ...
}
```

Once we've done that, linking to it is just like anything else: we got to Twig, we use the `path()` function, and everything is going to work perfectly:

```
{# app/Resources/views/Post/index.html.twig #}
{# ... #}

{% for post in posts %}
    <li>
        <a href="{{ path('post_show', { 'id': post.id }) }}">
            {{ post.title }}
        </a>
    </li>
{% endfor %}
```

Beyond the path and the `name` of the route, the only other common thing for routes is to add requirements. If you Google for `@Route` Symfony annotation, you'll find the documentation page that shows you how to add those. It's just another option on the `@Route` annotation:

```php
/**
 * @Route("/posts/{id}", name="post_show", requirements={"id"="\d+"})
 */
public function showAction(Post $post)
{
    // ...
}
```

And since this is all we really do with routes, it doesn't really get any messier than this.

Keep it Simple, Pass Along Feedback¶

And that's really it. With the `@Route` annotation and putting all of your templates in the same directory, your project already starts to get a lot smaller. So keep things simple, try this out, and let me know what you think.

Seeya next time :).