Create your First Page in Symfony

2.7 version

edit this page

Creating a new page - whether it's an HTML page or a JSON endpoint - is a simple two-step process:

1. Create a route: A route is the URL (e.g. `/about`) to your page and points to a controller;
2. Create a controller: A controller is the function you write that builds the page. You take the incoming request information and use it to create a Symfony `Response` object, which can hold HTML content, a JSON string or anything else.

Just like on the web, every interaction is initiated by an HTTP request. Your job is pure and simple: understand that request and return a response.

Creating a Page: Route and Controller ¶

fore continuing, make sure you've read the Installation chapter and can access your new Symfony app in the browser.

Suppose you want to create a page - `/lucky/number` - that generates a lucky (well, random) number and prints it. To do that, create a class and a method inside of it that will be executed when someone goes to`/lucky/number`:

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

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class LuckyController extends Controller
{
    /**
     * @Route("/lucky/number")
     */
    public function numberAction()
    {
        $number = rand(0, 100);
        return new Response(
            '<html><body>Lucky number: '.$number.'</body></html>'
        );
    }
}
```

Before diving into this, test it out!

> http://localhost:8000/app_dev.php/lucky/number

you setup a proper virtual host in Apache or Nginx, replace`http://localhost:8000` with your host name - like`http://symfony.dev/app_dev.php/lucky/number`.

If you see a lucky number being printed back to you, congratulations! But before you run off to play the lottery, check out how this works.

The `@Route` above `numberAction()` is called an annotation and it defines the URL pattern. You can also write routes in YAML (or other formats): read about this in the routing chapter. Actually, most routing examples in the docs have tabs that show you how each format looks.

The method below the annotation - `numberAction` - is called the controllerand is where you build the page. The only rule is that a controller mustreturn a Symfony Response object (and you'll even learn to bend this rule eventually).

hat's the app_dev.php in the URL?

Great question! By including `app_dev.php` in the URL, you're executing Symfony through a file - `web/app_dev.php` - that boots it in the `dev` environment. This enables great debugging tools and rebuilds cached files automatically. For production, you'll use clean URLs - like `http://localhost:8000/lucky/number` - that execute a different file - `app.php` - that's optimized for speed. To learn more about this and environments, see Environments.

Creating a JSON Response ¶

The `Response` object you return in your controller can contain HTML, JSON or even a binary file like an image or PDF. You can easily set HTTP headers or the status code.

Suppose you want to create a JSON endpoint that returns the lucky number. Just add a second method to `LuckyController`:

```php
// src/AppBundle/Controller/LuckyController.php
// ...

class LuckyController extends Controller
{
    // ...

    /**
     * @Route("/api/lucky/number")
     */
    public function apiNumberAction()
    {
        $data = array(
            'lucky_number' => rand(0, 100),
        );

        return new Response(
            json_encode($data),
            200,
            array('Content-Type' => 'application/json')
        );
    }
}
```

Try this out in your browser:

> http://localhost:8000/app_dev.php/api/lucky/number

You can even shorten this with the handy `JsonResponse`:

```php
// src/AppBundle/Controller/LuckyController.php
// ...

// --> don't forget this new use statement
use Symfony\Component\HttpFoundation\JsonResponse;

class LuckyController extends Controller
{
    // ...

    /**
     * @Route("/api/lucky/number")
     */
    public function apiNumberAction()
    {
        $data = array(
            'lucky_number' => rand(0, 100),
        );

        // calls json_encode and sets the Content-Type header
        return new JsonResponse($data);
    }
}
```

Dynamic URL Patterns: /lucky/number/{count} ¶

Woh, you're doing great! But Symfony's routing can do a lot more. Suppose now that you want a user to be able to go to `/lucky/number/5` to generate 5 lucky numbers at once. Update the route to have a `{wildcard}`part at the end:

Annotations YAML XML PHP

```
// src/AppBundle/Controller/LuckyController.php
// ...

class LuckyController extends Controller
{
    /**
     * @Route("/lucky/number/{count}")
     */
    public function numberAction()
    {
        // ...
    }

    // ...
}
```

Because of the `{count}` "placeholder", the URL to the page is different: it now works for URLs matching `/lucky/number/*` – for example`/lucky/number/5`. The best part is that you can access this value and use it in your controller:

```
// src/AppBundle/Controller/LuckyController.php
// ...

class LuckyController extends Controller
{

    /**
     * @Route("/lucky/number/{count}")
     */
    public function numberAction($count)
    {
        $numbers = array();
        for ($i = 0; $i < $count; $i++) {
            $numbers[] = rand(0, 100);
        }
        $numbersList = implode(', ', $numbers);

        return new Response(
            '<html><body>Lucky numbers: '.$numbersList.'</body></html>
        );
    }

    // ...
}
```

Try it by going to `/lucky/number/XX` – replacing XX with any number:

> http://localhost:8000/app_dev.php/lucky/number/7

You should see 7 lucky numbers printed out! You can get the value of any`{placeholder}` in your route by adding a `$placeholder` argument to your controller. Just make sure they have the same name.

The routing system can do a lot more, like supporting multiple placeholders (e.g. `/blog/{category}/{page})`), making placeholders optional and forcing placeholder to match a regular expression (e.g. so that `{count}` must be a number).

Find out about all of this and become a routing expert in the Routingchapter.

Rendering a Template (with the Service Container) ¶

If you're returning HTML from your controller, you'll probably want to render a template. Fortunately, Symfony comes with Twig: a templating language that's easy, powerful and actually quite fun.

So far, `LuckyController` doesn't extend any base class. The easiest way to use Twig – or many other tools in Symfony – is to extend Symfony's base`Controller` class:

```
// src/AppBundle/Controller/LuckyController.php
// ...

// --> add this new use statement
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class LuckyController extends Controller
{
    // ...
}
```

Using the templating Service ¶

This doesn't change anything, but it does give you access to Symfony'scontainer: an array-like object that gives you access to every useful object in the system. These useful objects are called services, and Symfony ships with a service object that can render Twig templates, another that can log messages and many more.

To render a Twig template, use a service called `templating`:

```
// src/AppBundle/Controller/LuckyController.php
// ...

class LuckyController extends Controller
{
    /**
     * @Route("/lucky/number/{count}")
     */
    public function numberAction($count)
    {
        // ...
        $numbersList = implode(', ', $numbers);

        $html = $this->container->get('templating')->render(
            'lucky/number.html.twig',
            array('luckyNumberList' => $numbersList)
        );

        return new Response($html);
    }

    // ...
}
```

You'll learn a lot more about the important "service container" as you keep reading. For now, you just need to know that it holds a lot of objects, and you can `get()` any object by using its nickname, like `templating` or `logger`. The `templating` service is an instance of `TwigEngine` and this has a `render()`method.

But this can get even easier! By extending the `Controller` class, you also get a lot of shortcut methods, like `render()`:

```
// src/AppBundle/Controller/LuckyController.php
// ...

/**
 * @Route("/lucky/number/{count}")
 */
public function numberAction($count)
{
    // ...

    /*
    $html = $this->container->get('templating')->render(
        'lucky/number.html.twig',
        array('luckyNumberList' => $numbersList)
    );

    return new Response($html);
    */

    // render: a shortcut that does the same as above
    return $this->render(
        'lucky/number.html.twig',
        array('luckyNumberList' => $numbersList)
    );
}
```

Learn more about these shortcut methods and how they work in theController chapter.

r more advanced users, you can also register your controllers as services.

Create the Template ¶

If you refresh now, you'll get an error:

> Unable to find template "lucky/number.html.twig"

Fix that by creating a new `app/Resources/views/lucky` directory and putting a `number.html.twig` file inside of it:

Twig PHP

```
{# app/Resources/views/lucky/number.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
    <h1>Lucky Numbers: {{ luckyNumberList }}</h1>
{% endblock %}
```

Welcome to Twig! This simple file already shows off the basics: like how the `{{ variableName }}` syntax is used to print something. The`luckyNumberList` is a variable that you're passing into the template from the `render` call in your controller.

The `{% extends 'base.html.twig' %}` points to a layout file that lives atapp/Resources/views/base.html.twig and came with your new project. It'sreally basic (an unstyled HTML structure) and it's yours to customize. The`{% block body %}` part uses Twig's inheritance system to put the content into the middle of the `base.html.twig` layout.

Refresh to see your template in action!

> http://localhost:8000/app_dev.php/lucky/number/9

If you view the source code, you now have a basic HTML structure thanks to `base.html.twig`.

This is just the surface of Twig's power. When you're ready to master its syntax, loop over arrays, render other templates and other cool things, read the Templating chapter.

Exploring the Project ¶

You've already created a flexible URL, rendered a template that uses inheritance and created a JSON endpoint. Nice!

It's time to explore and demystify the files in your project. You've already worked inside the two most important directories:

`app/`
Contains things like configuration and templates. Basically, anything that is not PHP code goes here.

`src/`
Your PHP code lives here.

99% of the time, you'll be working in `src/` (PHP files) or `app/` (everything else). As you get more advanced, you'll learn what can be done inside each of these.

The `app/` directory also holds a few other things, like the cache directory`app/cache/`, the logs directory `app/logs/` and `app/AppKernel.php`, which you'll use to enable new bundles (and one of a very short list of PHP files in `app/`).

The `src/` directory has just one directory – `src/AppBundle` – and everything lives inside of it. A bundle is like a "plugin" and you can find open source bundles and install them into your project. But even your code lives in a bundle – typically `AppBundle` (though there's nothing special about`AppBundle`). To find out more about bundles and why you might create multiple bundles (hint: sharing code between projects), see the Bundleschapter.

So what about the other directories in the project?

`vendor/`
: Vendor (i.e. third-party) libraries and bundles are downloaded here by the Composer package manager.

`web/`
: This is the document root for the project and contains any publicly accessible files, like CSS, images and the Symfony front controllers that execute the app (`app_dev.php` and `app.php`).

mfony is flexible. If you need to, you can easily override the default directory structure. See How to Override Symfony's default Directory Structure.

Application Configuration ¶

Symfony comes with several built-in bundles (open your`app/AppKernel.php` file) and you'll probably install more. The main configuration file for bundles is `app/config/config.yml`:

YAML XML PHP

```yaml
# app/config/config.yml
# ...

framework:
    secret: "%secret%"
    router:
        resource: "%kernel.root_dir%/config/routing.yml"
    # ...

twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"

# ...
```

The `framework` key configures FrameworkBundle, the `twig` key configures TwigBundle and so on. A lot of behavior in Symfony can be controlled just by changing one option in this configuration file. To find out how, see theConfiguration Reference section.

Or, to get a big example dump of all of the valid configuration under a key, use the handy `app/console` command:

```
$ app/console config:dump-reference framework
```

There's a lot more power behind Symfony's configuration system, including environments, imports and parameters. To learn all of it, see theConfiguration chapter.

What's Next? ¶

Congrats! You're already starting to master Symfony and learn a whole new way of building beautiful, functional, fast and maintainable apps.

Ok, time to finish mastering the fundamentals by reading these chapters:

- Controller
- Routing
- Creating and Using Templates

Then, in the Symfony Book, learn about the service container, the form system, using Doctrine (if you need to query a database) and more!

There's also a Cookbook packed with more advanced "how to" articles to solve a lot of problems.

Have fun!