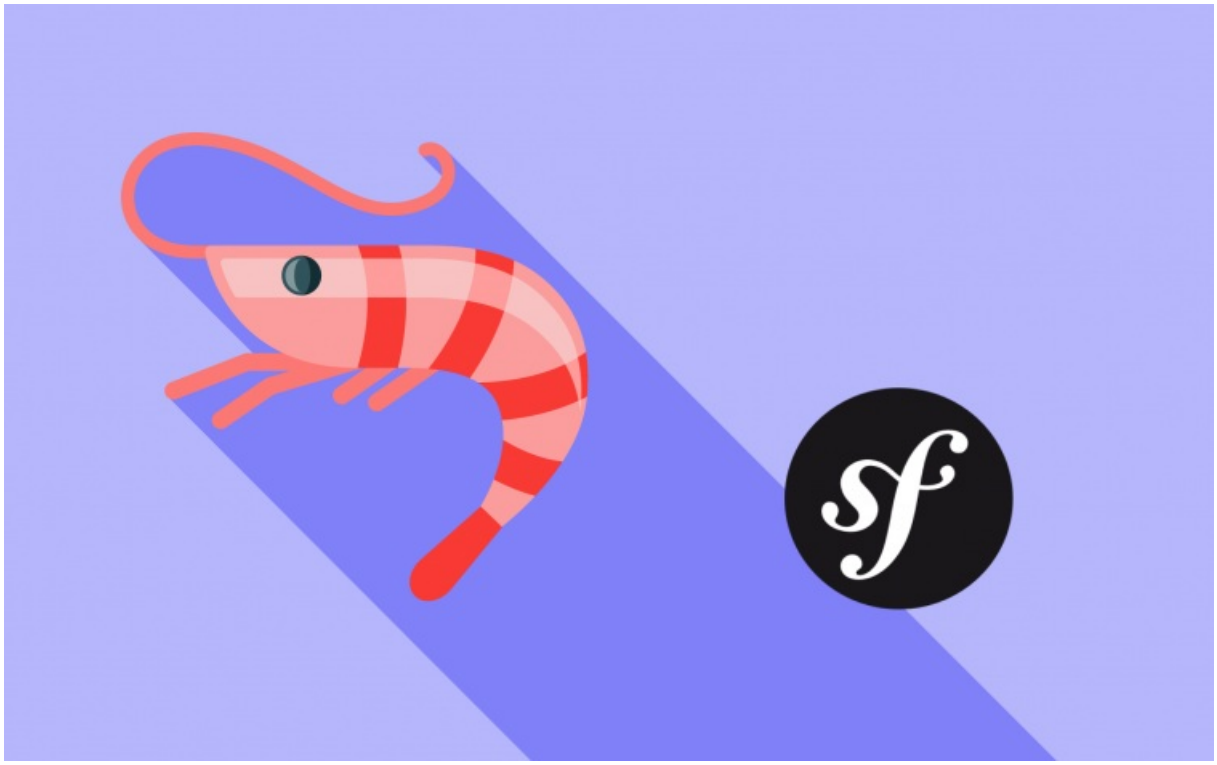
Joyful Development with Symfony 3



With <3 from SymfonyCasts

Chapter 1: Start Project

Well hey guys! You know what? I'm *pumped* that you're learning Symfony, because it's the hardest framework ever! Relax, I'm kidding. Symfony *does* have a reputation for being tough to learn, but this is a trap! Or at least, it's an outdated idea.

Look: Symfony can be *incredibly* simple and will put you in a position to write powerful, well-designed code, whether it's for an API or a traditional web app. And when it does get a bit more difficult, it's usually because you're learning best practices and object oriented goodness that's turning you into a better developer.

Symfony Components & Framework

So what is Symfony? First, it's a set of components: meaning PHP libraries. Actually, it's about 30 small libraries. That means that you could use Symfony in your non-Symfony project today by using one of its little libraries. One of my favorites is called Finder: it's really good at searching deep into directories for files.

But Symfony is also a framework where we've taken all of those components and glued them together for so that you can get things done faster. This series is all about doing *amazing* things with the Symfony framework.

The Symfony Installer

Let's get our first Symfony project rolling. Head over to Symfony.com and click 'Download'. Our first task is to get the Symfony Installer. Depending on your system, this means running commands from one of these boxes. Since I'm on a mac, I'll copy the curl command and paste it into the terminal:

```
sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
```

Copy the second command and run that to adjust some permissions:

```
sudo chmod a+x /usr/local/bin/symfony
```

This gives us a new symfony executable:

```
symfony
```

But hold on! This is *not* Symfony, it's the Symfony Installer: a tiny utility that makes it really easy to start new Symfony projects.

Downloading the Project

Let's start one! Run symfony new and then the name of the project. Call the project aqua_note:

```
symfony new aqua_note
```

I'll tell you more about it soon. In the background this is downloading a new Symfony project, unzipping it, making sure your system is configured, warning you of any problems and then dropping the new files into this aqua_note directory. Not bad!

Tip

The project name - aqua_note - is only used to determine the directory name: it's not important at all afterwards.

Move into the directory and check it out.

```
cd aqua_note
ls
```

This is *also* not Symfony: it's just a set of files and directories that form a web app that *use* the Symfony libraries. Those libraries - along with other third-party code - live in the vendor/ directory.

Before I explain the other directories, let's get this thing working! Run:

```
php bin/console server:run
```

to start the built in PHP web server. Yes, you can also use Nginx or Apache: but this is much easier for development. When you're done later, just hit Ctrl+C to stop the server.

As the comment says here, go to http://localhost:8000 in your browser. And boom! Congrats! This is your first page being executed by the Symfony framework. That's right: this is being rendered dynamically from the files inside of your project. At the bottom, you'll see one of the *best* features of Symfony: the web debug toolbar. This is *full* of debugging information - more on that later.

Ok, let's start building our own pages!

Chapter 2: Setup! PhpStorm + git

I've already opened the project in PhpStorm. It is by *far* the best editor for working with Symfony. And I'm not even getting paid to say this! Though, if there are any PhpStorm employees watching, I do accept payment in ice cream.

Anyways, it's awesome, but not free, but totally worth it. It has a free trial: so go download it and follow along with me.

The PhpStorm Symfony Plugin

To get *really* crazy, you'll want to install the *amazing*, incredible Symfony plugin. This thing makes Symfony development so absurdly fun, I'm going to walk you through its installation right now.

In Preferences, search for Symfony and click the plugins option. From here, click 'Browse Repositories` and then find the Symfony Plugin. You'll recognize it as the one with over 1.3 *million* downloads.

Tip

You should *also* find and install the PHP Annotations plugin. That will give you the awesome annotations auto-completion that you'll see in the video.

I already have it installed, but if you don't, you'll see an Install Plugin button. Click that and then restart PHPStorm. Once you're back, go into Preferences again and search for Symfony to find the new "Symfony Plugin" item. To activate the magic, click the "Enable Plugin for this Project" checkbox. Do this once per project. Oh, and also make sure that these paths say var/cache instead of app.

Go Deeper!

If you're interested in more PHPStorm tricks we have an entire screencast on it for you to enjoy.

Starting the git Repository

Ready to code? Wait! Before we break stuff, let's be good developers and start a new git repository. Our terminal is blocked by the built-in web server, so open up a new tab. Here, run:

```
git init
git add .
git status
```

The project already has a .gitignore file that's setup to avoid committing anything we don't want, like the vendor/ directory and the file that holds database credentials. Hey, thanks Symfony! Make the first commit and give it a clever message... hopefully, more clever than mine:

```
git commit
```

Chapter 3: First Page

Code goes in src/ and app/

You may have noticed that *most* of the committed files were in app/ and src/. That's on purpose: these are the *only* two directories you need to worry about. src/ will hold all the PHP classes you create and app/ will hold everything else: mostly configuration and template files. Ignore all the other directories for now and *just* focus on src/ and app/.

Building the First Page

Remember the functional homepage? It's coming from this DefaultController.php file. Delete that! Do it! Now we have an absolutely empty project. Refresh the homepage!

> No route found for "GET /"

Perfect! That's Symfony's way of saying "Yo! There's no page here."

Now back to the main event: building a real page.

Our top secret project is called AquaNote: a research database for Aquanauts. These cool underwater explorers log their discoveries of different sea creatures to this nautical site. Our first page will show details about a specific genus, for example, the octopus genus.

Creating a page in Symfony - or any modern framework - is two steps: a route and a controller. The route is a bit of configuration that says what the URL is. The controller is a function that builds that page.

Namespaces

So, step 1: create a route! Actually, we're going to start with step 2: you'll see why. Create a new class in AppBundle/Controller called GenusController. But wait! The namespace box is empty. That's ok, but PhpStorm can help us out a bit more. Hit escape and then right-click on src and select "mark directory as sources root".

Now re-create GenusController. This time it fills in the namespace for me:

18 lines src/AppBundle/Controller/GenusController.php

```
<?php
namespace AppBundle\Controller;
... lines 4 - 7
class GenusController
{
... lines 10 - 16
}
```

Go Deeper!

If namespaces are new to you, welcome! Take a break and watch our PHP Namespaces Tutorial.

The most important thing is that the namespace *must* match the directory structure. If it doesn't, Symfony won't be able to find the class. By setting the sources root, PhpStorm is able to guess the namespace. And that saves us precious time.

Controller and Route

Inside, add a public function showAction():

18 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 7
class GenusController
{
... lines 10 - 12
public function showAction()
{
... line 15
}
}
```

Hey, this is the controller - the function that will (eventually) build the page - and its name isn't important. To create the route, we'll use annotations: a comment that is parsed as configuration. Start with /** and add @Route. Be sure to let PhpStorm autocomplete that from the FrameworkExtraBundle by hitting tab. This is important: it added a use statement at the top of the class that we need. Finish this by adding "/genus":

18 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
... lines 6 - 7
class GenusController
{
/**
* @Route("/genus")
*/
public function showAction()
... lines 14 - 16
}
```

Beautiful, that's the route and the URL for the page is /genus.

Returning a Response

As I already said: the controller is the function right below this, and its job is to build the page. The *only* rule for a controller is that it must return a Symfony Response object.

But hold on. Let's just all remember what our *only* job is as web developers: to understand the incoming request and send back a response, whether that's an HTML response, a JSON response of a PDF file. Symfony is modeled around this idea.

Keep things simple: return new Response. The Response class is the one from the HttpFoundation component. Hit tab to auto-complete it. This adds the use statement on top that we need. For the content, how about: 'Under the Sea!':

18 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 5
use Symfony\Component\HttpFoundation\Response;
class GenusController
{
/**
* @Route("/genus")
*/
public function showAction()
{
return new Response('Under the sea!');
}
}
```

That's it!

We've only created one file with one function, but we already have a route, a controller and a lot of sea floor that needs discovering!

If you refresh the homepage, well... that's not going to work. Navigate instead to the URL: /genus. Woh! There's your first page in Symfony, done in about 10 lines of code. Simple enough for you?

Next, let's create a dynamic URL.

Chapter 4: Routing Wildcards

This page has a boring, hardcoded URL. What our aquanauts deserve is a dynamic route that can handle the URL for any genus - like /genus/octopus or /genus/hippocampus which is the genus that sea horses belong to. Oh man, sea horses are cute.

How? Change the URL to /genus/{genusName}:

18 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 7
class GenusController
{
/**
* @Route("/genus/{genusName}")
*/
public function showAction($genusName)
... lines 14 - 16
}
```

This genusName part could be named anything: the important part is that it is surrounded by curly braces. As soon as you do this, you are allowed to have a $genusName argument to your controller. When we go to /genus/octopus this variable will be set to octopus. That's pretty awesome.

The important thing is that the routing wildcard matches the variable name.

To test this is, change the message in the response to 'The genus: '.$genusName:

18 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 7
class GenusController
{
/**
* @Route("/genus/{genusName}")
*/
public function showAction($genusName)
{
return new Response('The genus: '.$genusName);
}
}
```

Tip

Be careful when rendering direct user input (like we are here)! It introduces a security issue called XSS - read more about XSS here.

Head back to the browser and refresh. Ah! A 404 error. That's because the URL is no longer /genus, it's now /genus/something: we have to have something on the other side of the URL. Throw octopus on the end of that (/genus/octopus). There's the new message. And of course, we could change this to whatever we want.

So what would happen if the wildcard and the variable name *didn't* match? Um I don't know: let's try it: change the wildcard name and refresh!

18 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 7
class GenusController
{
/**
* @Route("/genus/{genusName2}")
*/
public function showAction($genusName)
... lines 14 - 16
}
```

OMG: that's a sweet error:

> The Controller::showAction() requires that you provide a value for the $genusName argument.

What Symfony is trying to tell you is:

> Hey fellow ocean explorer, I'm trying to call your showAction(), but I can't figure out what value to pass to genusName because I don't see a {genusName} wildcard in the route. I'm shore you can help me.

As long as those always match, you'll be great.

Listing all Routes

When you load this page, Symfony loops over *all* the routes in your system and asks them one-by-one: do you match /genus/octopus? Do *you* match /genus/octopus? As soon as it finds *one* route that matches that URL, it stops and calls that controller.

So far, we only have one route, but eventually we'll have a lot, organized across many files. It would be *swimming* if we could get a big list of every route. Ha! We can!

Symfony comes with an awesome debugging tool called the console. To use it, go to the terminal and run

php bin/console

This returns a big list of commands that you can run. Most of these help you with debugging, some generate code and others do things like clear caches. We're interested in debug:router. Let's run that:

php bin/console debug:router

Nice! This prints out *every* route. You can see our route at the bottom: /genus/{genusName}. But there are other routes, I wonder where those are coming from? Those routes give you some debugging tools - like the little web debug toolbar we saw earlier. I'll show you where these are coming from later.

When we add more routes later, they'll show up here too.

Ok, fun fact! A baby seahorse is called a "fry".

How about a relevant fun fact? You now know 50% of Symfony. Was that really hard? The routing-controller-response flow is the first half of Symfony, and we've got it crossed off.

Now, let's dive into the second half.

Chapter 5: Intro to Services

Ok! The first half of Symfony: route-controller-response is in the books!

The second half is all about useful objects. Obviously, returning a string response like this is not going to take us very far. Our aquanauts demand more! In real life, we might need to render a template, query a database, or even turn objects into JSON for an API!

Symfony comes with many, optional, useful objects to help out with stuff like this. For example, want to send an email? Symfony has an object for that. How about log something? There's an object for that.

These objects are commonly called services, and that's important: when you hear the word service, just think "useful object".

Service Container

To keep track of all of these services, Symfony puts them into one big associative array called the container. Each object has a key - like mailer or logger. And to be more honest with you - sorry, I do like to lie temporarily - the container is actually an object. But think of it like an array: each useful object has an associated key. If I give you the container, you can ask for the logger service and it'll give you that object.

The second half of Symfony is all about finding out what objects are available and how to use them. Heck, we'll even add our *own* service objects to the container before too long. That's when things get really cool.

Accessing the Container

The first useful object is the templating service: it renders Twig templates. To get access to the service container, you need to extend Symfony's base controller.

Go Deeper!

Why does extending Controller give you access to the container? Find out: Injecting the Container: ContainerAwareInterface (advanced).

In GenusController, add extends Controller from FrameworkBundle. Hit tab to autocomplete and get the use statement:

24 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 5
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
... lines 7 - 8
class GenusController extends Controller
{
... lines 11 - 22
}
```

To get the templating service, add $templating = $this->container->get('templating'):

24 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 8
class GenusController extends Controller
{
... lines 11 - 13
public function showAction($genusName)
{
$templating = $this->container->get('templating');
... lines 17 - 21
}
}
```

The container pretty much only has one method: get. Give it the nickname to the service and it will return that object. It's super simple.

Quickly, open the var directory, right click on the cache directory and click "mark this directory as excluded". Symfony caches things... that's not important yet, but excluding this *is* important: this directory confuses autocompletion.

Now type $this->container->get('templating'). Well hey autocompletion!

Rendering a Template

With the templating object we can... well... render a template! Add $html = $templating->render('') followed by the name of the template. This could be anything, but let's be logical: genus/show.html.twig. I'll show you where this lives in a second:

24 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 8
class GenusController extends Controller
{
... lines 11 - 13
public function showAction($genusName)
{
$templating = $this->container->get('templating');
$html = $templating->render('genus/show.html.twig', array(
'name' => $genusName
));
... lines 20 - 21
}
}
```

We'll also want to pass some variables into the template. Pass a name variable into Twig that's set to $genusName.

Finally, what do we always do in Symfony controllers? We always return a Symfony's Response object. Stick, that $html into the response object and return it:

24 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 8
class GenusController extends Controller
{
... lines 11 - 13
public function showAction($genusName)
{
... lines 16 - 20
return new Response($html);
}
}
```

Go Deeper!

You can actually return *anything* from a controller via the kernel.view event: The kernel.view Event (advanced)

Create the Template

Ok, where do templates live? Ah, it's so simple: templates live in app/Resources/views. The one we're looking for will be in app/Resources/views/genus/show.html.twig. The existing index.html.twig template was for the original homepage. Check it out if you want to, then delete it!

Create a new genus directory and then a new file: show.html.twig. Welcome to Twig! You'll love it. Add an <h1> tag with The Genus and then {{ name }} to print the name variable. More on Twig in a second:

2 lines app/Resources/views/genus/show.html.twig

```
<h1>The Genus {{ name }}</h1>
```

But that's it! Refresh the browser. Check out that sweet h1 tag.

Now back up: we just did something really cool: used our first service. We now know that rendering a template isn't done by some deep, dark part of Symfony: it's done by the templating object. In fact, Symfony doesn't really do *anything*: everything is done by one of these services.

Chapter 6: Listing and Using Services

Rendering a template is pretty common, so there's a shortcut when you're in a controller. Replace all of this code with a simple return $this->render:

21 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 8
class GenusController extends Controller
{
... lines 11 - 13
public function showAction($genusName)
{
return $this->render('genus/show.html.twig', array(
'name' => $genusName
));
}
}
```

That's it. Make sure this works by refreshing.

So what does this magic-looking render() function actually do? Let's find out! Hold command or control (depending on your OS) and click render() to be taken straight into the base Controller class where this function lives: deep in the heart of Symfony:

398 lines vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

```
... lines 1 - 38
abstract class Controller implements ContainerAwareInterface
{
... lines 41 - 185
protected function render($view, array $parameters = array(), Response $response = null)
{
if ($this->container->has('templating')) {
return $this->container->get('templating')->renderResponse($view, $parameters, $response);
}
... lines 191 - 202
}
... lines 204 - 396
}
```

Ah, hah! In reality, this function simply goes out to the templating service - just like we did - and calls a method named renderResponse(). This method is like the render() function we called, except that it wraps the HTML in a Response object for convenience.

Here's the point: the base Controller class has a lot of shortcut methods that you'll use. But behind the scenes, these don't activate some weird, core functionality in Symfony. Instead, everything is done by one of the services in the container. Symfony doesn't really *do* anything: all the work is done by different services. That's awesome.

What Services are there?

Oh, you want to know what *other* services are hiding in the container? Me too! To find that out, head back to the terminal and use the handy console:

php bin/console

Check out that debug:container command - run that!

php bin/console debug:container

You should see a short list of service. Ah, I mean, you should see over *200* useful objects in Symfony that you get access to out of the box. But don't worry about memorizing these: as you use Symfony, you'll find out which ones are important to you and your project by reading the docs on how to accomplish different things.

But sometimes, you can just guess! For example, does Symfony have a service for logging? Um, maybe?! We could Google that, or we could pass an argument to this command to search for services matching "log":

php bin/console debug:container log

Wow, there are 18! Here's a secret: the service you usually want is the one with the shortest name. In this case: logger. So if you wanted to log something, just grab this out of the container and use it. This command also shows you what *class* you'll get back, which you can use to find the methods on it.

We just figured this out without *any* documentation.

There's a lot more to say later about services and the container. In fact, it's one of the most fundamentally important things that makes Symfony so special... and so fast.

Chapter 7: Twig: For a Good time with Templates

Unless you're building a pure API, Twig is your new best friend. Rarely do you find a library that's this much fun to use. It's also really easy, so let me just give you a quick intro.

{{ SaySomething }}, {% doSomething %}

Twig has two syntaxes: {{ }} - which is the "say something" tag - and {% %} - which is the "do something" tag. If you're printing something, you aways write {{ then a variable name, a string or any expression: Twig looks a lot like JavaScript.

But if you're writing code that *won't* print something - like an if statement a for loop, or setting a variable, you'll use {% %}.

Head over to Twig's website at twig.sensiolabs.org, click Documentation and scroll down. Ah, this is a list of *everything* Twig does.

Look at the Tags column first: this is the short list of *all* "do something" tags. Click on if to see some usage. Do something tags are always {% and then if or set or one of these "tags". The for tag - used as {% for %} is for looping. You'll probably end up only using 5 or 6 of these commonly.

Twig also has other things like functions... which are exactly like functions in every language ever. Filters are a bit more interesting: check out lower. Really, these are functions, but with a trendier syntax: just print something, then use the pipe (|) to pass that value into a filter. You can have filter after filter. And, you can create your own.

The dump() Function

Let's make some magic happen in our Twig template. Our Aquanauts will take notes about each genus, and those will render on this page. Create a cool $notes variable with some hardcoded text and pass it into our Twig template:

28 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 8
class GenusController extends Controller
{
... lines 11 - 13
public function showAction($genusName)
{
$notes = [
'Octopus asked me a riddle, outsmarted me',
'I counted 8 legs... as they wrapped around me',
'Inked!'
];
return $this->render('genus/show.html.twig', array(
'name' => $genusName,
'notes' => $notes
));
}
}
```

But before we loop over this, I want to show you a small piece of awesome: the dump() function:

3 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 2
{{ dump() }}
```

This is like var_dump() in PHP, but better, *and* you can use it without *any* arguments to print details about *every* available variable.

Refresh the browser! There's the name variable, notes and a bonus: app - a global variable that Symfony adds to every template. More on that in the future. With the dump() function, you can expand the variables in really cool ways. Oh, and bonus time: you can also use dump() in PHP code: Symfony gives us that function.

Go Deeper!

See more usage examples of the dump() function in The dump Function for Debugging of the separate Twig screencast. chapter.

The for Tag

To print out the notes, add a <ul> and open up a for tag with {% for note in notes %}. Close it with an {% endfor %} tag. Now, it's simple: print out each note, which is a string:

8 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 2
<ul>
{% for note in notes %}
<li>{{ note }}</li>
{% endfor %}
</ul>
```

Back to the browser to see what we've got. Refresh! Well, it's not pretty yet, but it is working. Open the source: it's still *just* this html, there's no HTML layout. Time to fix that.

Chapter 8: Twig Layouts (Template Inheritance)

To get a layout, add a new do something tag at the top of show.html.twig: extends 'base.html.twig':

10 lines app/Resources/views/genus/show.html.twig

```
{% extends 'base.html.twig' %}
<h1>The Genus {{ name }}</h1>
... lines 4 - 10
```

This says that we want base.html.twig to be our base template. But where does that file live? Remember: all templates live in app/Resources/views. And look, there's base.html.twig. This little file actually came with Symfony and it's your's to customize.

Refresh the browser after just this small change. Nice, a huge error

> A template that extends another one can't have a body...

So what does that mean?

In Twig, layouts work via template inheritance. Ooooh. The base.html.twig template is filled with blocks:

14 lines app/Resources/views/base.html.twig

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>{% block title %}Welcome!{% endblock %}</title>
{% block stylesheets %}{% endblock %}
<link rel="icon" type="image/x-icon" href="{{ asset('favicon.ico') }}" />
</head>
<body>
{% block body %}{% endblock %}
{% block javascripts %}{% endblock %}
</body>
</html>
```

The job of the child template is to define content that should go into each of these blocks. For example, if you want your child template - show.html.twig in this case - to put content inside this body block, you need to *override* it. If you want to replace the title, then you'll override the title block.

Right now, our show.html.twig file is just barfing content. We're telling Twig we want to use base.html.twig, but it doesn't know *where* in that file this content should be placed.

To fix this, wrap *all* of the content in a block: {% block body %}. At the end, close it with {% endblock %}:

11 lines app/Resources/views/genus/show.html.twig

```
{% extends 'base.html.twig' %}
{% block body %}
<h1>The Genus {{ name }}</h1>
<ul>
{% for note in notes %}
<li>{{ note }}</li>
{% endfor %}
</ul>
{% endblock %}
```

Oh, and the names of these blocks are not important at all. You can change them to whatever you want and can add as many as you need.

The Web Debug Toolbar and Profiler

With this fixed up, head back to the browser and refresh. Cool! It's the same page, but now it has a full html source. Bonus time! Once you have a full html page, the web debug toolbar makes an appearance. This is a killer feature in Symfony: it includes information about which route was matched, which controller was executed, how fast the page loaded, who is logged in and more.

You can also click any of the icons to get even more detailed information in the profiler, including this amazing timeline that shows you exactly how long each part of your application took to render. This is *amazing* for debugging and profiling. There's also details in here on Twig, security, routes and other cool stuff. We'll keep exploring this as we go along.

Overriding a Second Block

Ok, the title of the page - "welcome" - well, that's not terribly inspiring or accurate for this page. That comes from the base layout, but it's wrapped in a block called title. Let's override that!

Add {% block title %}Genus {{ name }}{% endblock %}:

13 lines app/Resources/views/genus/show.html.twig

```
{% extends 'base.html.twig' %}
{% block title %}Genus {{ name }}{% endblock %}
... lines 4 - 13
```

The order of blocks doesn't matter: this could be above or below the body. Back to the browser and refresh! Ah ha! There's our new title -- not too shabby. That's it for Twig -- what's not to love?

Go Deeper!

If you want more, we have a whole screencast on *just* Twig templating engine: Twig Templating for Friendly Frontend Devs.

Chapter 9: Loading CSS & JS Assets

We have an HTML layout, yay! Good for us!

But... it's *super* boring... and that's just no fun. Besides, I want to talk about assets!

If you download the code for this project, in the start/ directory, you'll find a tutorial/ directory. I've already copied this into my project. It holds some goodies that we need to help get things looking less ugly, and more interesting!

Copying web files

First, copy the 4 directories in web/... to web/: this includes some CSS, images, JS and vendor files for Bootstrap and FontAwesome:

web/css
web/images
web/js
web/vendor

These are boring, normal, traditional static files: we're not doing anything fancy with frontend assets in this screencast.

Go Deeper!

One way to get fancy is by using Gulp to process, minify and combine assets. See Gulp! Refreshment for Your Frontend Assets.

Ok, important thing: the web/ directory is the *document root*. In other words, anything in web/ can be accessed by the public. If I wanted to load up that favicon.ico, I'd use my hostname /favicon.ico - like http://localhost:8000/favicon.ico. If a file is *outside* of web, then it's *not* publicly accessible.

Including Static Assets

Ok, more work: go into the app/ directory and copy the new base.html.twig file. Paste that over the original and open it up:

41 lines app/Resources/views/base.html.twig

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>{% block title %}AquaNote!{% endblock %}</title>
{% block stylesheets %}
<link rel="stylesheet" href="{{ asset('vendor/bootstrap/css/bootstrap.min.css') }}">
<link rel="stylesheet" href="{{ asset('css/styles.css') }}">
<link rel="stylesheet" href="{{ asset('vendor/fontawesome/css/font-awesome.min.css') }}">
{% endblock %}
<link rel="icon" type="image/x-icon" href="{{ asset('favicon.ico') }}" />
</head>
<body>
<div class="search-bar">
<form method="GET" action="" class="js-sea-search sea-search">
<input type="search" name="q" placeholder="Search Sea Creatures" autocomplete="off" class="search-input">
</form>
</div>
<header class="header">
<img class="logo-icon" src="{{ asset('images/aquanote-logo.png') }}">
<h1 class="logo">AquaNote</h1>
<ul class="navi">
<li class="search"><a href="#" class="js-header-search-toggle"><i class="fa fa-search"></i></a></li>
<li><a href="#">Login</a></li>
</ul>
</header>
{% block body %}{% endblock %}
<div class="footer">
<p class="footer-text">Made with <span class="heart"><3</span> <a
href="https://knpuniversity.com">KnpUniversity</a></p>
</div>
{% block javascripts %}
<script src="//code.jquery.com/jquery-2.1.4.min.js"></script>
<script src="{{ asset('js/main.js') }}"></script>
{% endblock %}
</body>
</html>
```

Hey! We have some real-looking HTML! To bring this to life, we need to include some of those CSS and JS assets that we just put into web/. And here's the key: Symfony doesn't care about your assets... at all. It's not personal, it keeps things simple. You include CSS and JS files the way you always have: with tried-and-true link and script tags. These paths are relative to the web/ directory, because that's the document root.

The stylesheets and javascripts blocks

Ok ok, in reality there are *two* little-itty-bitty Symfony things to show you about assets. First, notice that the link tags live inside a block called stylesheets:

41 lines app/Resources/views/base.html.twig

```
... lines 1 - 6
{% block stylesheets %}
<link rel="stylesheet" href="{{ asset('vendor/bootstrap/css/bootstrap.min.css') }}">
<link rel="stylesheet" href="{{ asset('css/styles.css') }}">
<link rel="stylesheet" href="{{ asset('vendor/fontawesome/css/font-awesome.min.css') }}">
{% endblock %}
... lines 12 - 41
```

Really, technically, that does... nothing! Seriously: you don't have to do this, it will make no difference... for now.

But, in the future, doing this will give you the power to add page-specific CSS by adding *more* link tags to the bottom of the stylesheets block from inside a child template. I'll show you that later. Just know that it's a good practice to put CSS inside of a block, like stylesheets.

Go Deeper!

How can you add page-specific CSS or JS files? See ReactJS talks to your API.

The same is true for script tags: I've got mine in a block called javascripts:

41 lines app/Resources/views/base.html.twig

```
... lines 1 - 34
{% block javascripts %}
<script src="//code.jquery.com/jquery-2.1.4.min.js"></script>
<script src="{{ asset('js/main.js') }}"></script>
{% endblock %}
... lines 39 - 41
```

The asset function

You're probably already looking at the *second* important Symfony thing about assets: the asset() function. *Whenever* you refer to a static file, you'll wrap the path in {{ asset() }}. This does... you guessed it! Nothing! Ok, that's not totally true. But it really doesn't do much, and you'd be just fine if you forgot it and hardcoded the path.

So what *does* asset() do? Well, if you eventually deploy and use a CDN, it will save your butt. With just one tiny config change, Symfony can prefix *every* static URL with your CDN host. So /css/styles.css becomes http://superfastcdn.com/css/styles.css. That's pretty awesome, so be good and use asset() in case you need it. You can also do some cool cache-busting stuff.

Other than the asset stuff, the base layout is just like before: it has a title block, a body block in the middle and some javascripts. We just added the pretty markup.

Updating show.html.twig

Let's finish this! Copy show.html.twig and overwrite our boring version:

39 lines app/Resources/views/genus/show.html.twig

```
{% extends 'base.html.twig' %}
{% block title %}Genus {{ name }}{% endblock %}
{% block body %}
<h2 class="genus-name">{{ name }}</h2>
<div class="sea-creature-container">
<div class="genus-photo"></div>
<div class="genus-details">
<dl class="genus-details-list">
<dt>Subfamily:</dt>
<dd>Octopodinae</dd>
<dt>Known Species:</dt>
<dd>289</dd>
<dt>Fun Fact:</dt>
<dd>Octopuses can change the color of their body in just three-tenths of a second!</dd>
</dl>
</div>
</div>
<div class="notes-container">
<h2 class="notes-header">Notes</h2>
<div><i class="fa fa-plus plus-btn"></i></div>
</div>
<section id="cd-timeline">
{% for note in notes %}
<div class="cd-timeline-block">
<div class="cd-timeline-img">
<img src="{{ asset('images/leanna.jpeg') }}" class="img-circle" alt="Leanna!">
</div>
<div class="cd-timeline-content">
<h2><a href="#">AquaPelham</a></h2>
<p>{{ note }}</p>
<span class="cd-date">Dec. 10, 2015</span>
</div>
</div>
{% endfor %}
</section>
{% endblock %}
```

And yep, it's also similar to before - I swear I'm not trying to sneak in any magic! It still extends base.html.twig, prints out the genus name and loops over the notes. Oh, and hey! When I refer to the image - which is a static file - I'm using the asset() function.

Ok, ready for this? Refresh the page. Boom! So much prettier.

These days, you can do some pretty crazy things with assets via frontend tools like Gulp or PHP tools like Assetic. But you *might* not need any of these. If you can, keep it simple.

Chapter 10: JSON Responses + Route Generation

Okay, this is cool... but what about APIs and JavaScript frontends and all that new fancy stuff? How does Symfony stand up to that? Actually, it stands up *wonderfully*: Symfony is a first-class tool for building APIs. Seriously, you're going to love it.

Since the world is now a mix of traditional apps that return HTML and API's that feed a JavaScript frontend, we'll make an app that's a mixture of both.

Right now, the notes are rendered server-side inside of the show.html.twig template. But that's not awesome enough! If an aquanaut adds a new comment, I need to see it instantly, *without* refreshing. To do that, we'll need an API endpoint that returns the notes as JSON. Once we have that, we can use JavaScript to use that endpoint and do the rendering.

Creating API Endpoints

So how *do* you create API endpoints in Symfony? Ok, do you remember what a controller *always* returns? Yes, a Response! And ya know what? Symfony doesn't care whether that holds HTML, JSON, or a CSV of octopus research data. So actually, this turns out to be really easy.

Create a new controller: I'll call it getNotesAction(). This will return notes for a specific genus. Use @Route("/genus/{genusName}/notes"). We really only want this endpoint to be used for GET requests to this URL. Add @Method("GET"):

40 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 9
class GenusController extends Controller
{
... lines 12 - 21
/**
* @Route("/genus/{genusName}/notes")
* @Method("GET")
*/
public function getNotesAction($genusName)
{
... lines 28 - 37
}
}
```

Without this, the route will match a request using *any* HTTP method, like POST. But with this, the route will only match if you make a GET request to this URL. Did we need to do this? Well no: but it's pretty trendy in API's to think about which HTTP method should be used for each route.

Missing Annotation use Statement

Hmm, it's highlighting the @Method as a missing import. Ah! Don't forget when you use annotations, let PhpStorm autocomplete them for you. That's important because when you do that, PhpStorm adds a use statement at the top of the file that you need:

40 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
... lines 6 - 40
```

If you forget this, you'll get a pretty clear error about it.

Ok, let's see if Symfony sees the route! Head to the console and run debug:router:

php bin/console debug:router

Hey! There's the new route at the bottom, with its method set to GET.

The JSON Controller

Remove the $notes from the other controller: we won't pass that to the template anymore:

40 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 9
class GenusController extends Controller
{
... lines 12 - 14
public function showAction($genusName)
{
return $this->render('genus/show.html.twig', array(
'name' => $genusName,
));
}
... lines 21 - 38
}
```

In the new controller, I'll paste a new $notes variable set to some beautiful data:

40 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 9
class GenusController extends Controller
{
... lines 12 - 25
public function getNotesAction($genusName)
{
$notes = [
['id' => 1, 'username' => 'AquaPelham', 'avatarUri' => '/images/leanna.jpeg', 'note' => 'Octopus asked me a riddle, outsmarted me', 'date' => 'Dec. 10, 2015'],
['id' => 2, 'username' => 'AquaWeaver', 'avatarUri' => '/images/ryan.jpeg', 'note' => 'I counted 8 legs... as they wrapped around me', 'date' => 'Dec. 1, 2015'],
['id' => 3, 'username' => 'AquaPelham', 'avatarUri' => '/images/leanna.jpeg', 'note' => 'Inked!', 'date' => 'Aug. 20, 2015'],
];
... lines 33 - 37
}
}
```

We're not using a database yet, but you can already see that this kind of *looks* like it came from one: it has a username, a photo for each avatar, and the actual note. It'll be pretty easy to make this dynamic in the next episode.

Next, create a $data variable, set it to an array, and put the $notes in a notes key inside of that. Don't worry about this: I'm just creating a future JSON structure I like:

40 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 27
$notes = [
['id' => 1, 'username' => 'AquaPelham', 'avatarUri' => '/images/leanna.jpeg', 'note' => 'Octopus asked me a riddle, outsmarted me', 'date' => 'Dec. 10, 2015'],
['id' => 2, 'username' => 'AquaWeaver', 'avatarUri' => '/images/ryan.jpeg', 'note' => 'I counted 8 legs... as they wrapped around me', 'date' => 'Dec. 1, 2015'],
['id' => 3, 'username' => 'AquaPelham', 'avatarUri' => '/images/leanna.jpeg', 'note' => 'Inked!', 'date' => 'Aug. 20, 2015'],
];
$data = [
'notes' => $notes
];
... lines 36 - 40
```

Now, how do we finally return $data as JSON? Simple: return new Response() and pass it json_encode($data):

40 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 9
class GenusController extends Controller
{
... lines 12 - 25
public function getNotesAction($genusName)
{
... lines 28 - 36
return new Response(json_encode($data));
}
}
```

Simple!

Hey, let's see if this works. Copy the existing URL and add /notes at the end. Congratulations, you've just created your first Symfony API endpoint.

JsonResponse

But you know, that *could* have been easier. Replace the Response with new JsonResponse and pass it $data without the json_encode:

41 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 7
use Symfony\Component\HttpFoundation\JsonResponse;
... lines 9 - 10
class GenusController extends Controller
{
... lines 13 - 26
public function getNotesAction($genusName)
{
... lines 29 - 37
return new JsonResponse($data);
}
}
```

This does two things. First, it calls json_encode() for you. Hey thanks! And second, it sets the application/json Content-Type header on the Response, which we could have set manually, but this is easier.

Refresh. It still works perfectly.

Chapter 11: Generating URLs

We now have *two* pages: the HTML /genus/{genusName} page and the JSON endpoint. Ok ok, the JSON endpoint isn't really a *page*, at least not in the traditional sense. But pretend it is for a second. Pretend that we want to link from the HTML page to the JSON endpoint so the user can see it. Yes yes, we're going to do something fancier with JavaScript in a minute, but stay with me: this is important.

In show.html.twig, get rid of all this notes stuff because we're not passing in a notes variable anymore. Instead, near the top, add a new anchor tag. I want to put the URL to the getNotesAction page. Fill this in with /genus/{{ name }}/notes.

Perfect right! Wrong! Ok, only kind of wrong. This *will* work: you can click the link and go that URL. But this kinda sucks: if I decided that I needed to change the URL for this route, we would need to hunt down and update *every* link on the site. That's ridiculous.

Instead, routing has a second purpose: the ability to generate the URL to a specific route. But to do that, you need to give the route a unique name. After the URL, add comma and name="genus_show_notes":

41 lines src/AppBundle/Controller/GenusController.php

```
... lines 1 - 10
class GenusController extends Controller
{
... lines 13 - 22
/**
* @Route("/genus/{genusName}/notes", name="genus_show_notes")
* @Method("GET")
*/
public function getNotesAction($genusName)
{
... lines 29 - 38
}
}
```

The name can be anything, but it's usually underscored and lowercased.

The Twig path() Function

To generate a URL in Twig, use the path() function. This has two arguments. The first is the name of the route - genus_show_notes. The second, which is optional, is an associative array. In Twig, an associative array is written using { }, just like JavaScript or JSON. Here, pass in the values for any wildcards that are in the route. This route has {genusName}, so pass it genusName set to the name variable:

29 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
<h2 class="genus-name">{{ name }}</h2>
<a href="{{ path('genus_show_notes', {'genusName': name}) }}">Json Notes</a>
... lines 9 - 27
{% endblock %}
```

Ok, go back and refresh! This generates the same URL... so that might seem kind of boring. But if you ever need to change the URL for the route, all the links would automatically update. When you're moving fast to build something amazing, that's huge.

Linking to a JSON endpoint isn't realistic. Let's do what we originally planned: use JavaScript to give us a dynamic frontend.

Remove the link. In base.html.twig, we already have a few JavaScript files that are included on every page. But now, I want to include some JavaScript on *just* this page - I don't need this stuff everywhere.

Page-Specific JavaScript (or CSS)

Remember from earlier that those script tags live in a javascripts block. Hey, that's perfect! In the child template, we can override that block: {% block javascripts %} then {% endblock %}:

38 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 23
{% block javascripts %}
... lines 25 - 36
{% endblock %}
```

Now, whatever JS we put here will end up at the bottom of the layout. Perfect, right?

No, not perfect! When you override blocks, you override them *completely*. With this code, it will completely replace the other scripts in the base template. I don't want that! I really want to *append* content to this block.

The secret awesome solution to this is the parent() function:

38 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 23
{% block javascripts %}
{{ parent() }}
... lines 26 - 36
{% endblock %}
```

This prints all of the content from the parent block, and *then* we can put our cool stuff below that.

Including the ReactJS Code

Here's the goal: add some JavaScript that will make an AJAX request to the notes API endpoint and use that to render them with the same markup we had before. We'll use ReactJS to do this. It's powerful... and super fun, but if it's new to you, don't worry. We're not going to learn it now, just preview it to see how to get our API working with a JavaScript frontend.

First, include three external script tags for React itself. Next, I'm going to include one more script tag that points to a file in *our* project: notes.react.js:

38 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 23
{% block javascripts %}
{{ parent() }}
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-dom.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
<script type="text/babel" src="{{ asset('js/notes.react.js') }}"></script>
... lines 31 - 36
{% endblock %}
```

Let's check that file out! Remember, it's in web/js/notes.react.js:

69 lines web/js/notes.react.js

```
var NoteSection = React.createClass({
getInitialState: function() {
return {
notes: []
}
},
componentDidMount: function() {
this.loadNotesFromServer();
setInterval(this.loadNotesFromServer, 2000);
},
loadNotesFromServer: function() {
$.ajax({
url: '/genus/octopus/notes',
success: function (data) {
this.setState({notes: data.notes});
}.bind(this)
});
},
render: function() {
return (
<div>
<div className="notes-container">
<h2 className="notes-header">Notes</h2>
<div><i className="fa fa-plus plus-btn"></i></div>
</div>
<NoteList notes={this.state.notes} />
</div>
);
}
});
var NoteList = React.createClass({
render: function() {
var noteNodes = this.props.notes.map(function(note) {
return (
<NoteBox username={note.username} avatarUri={note.avatarUri} date={note.date} key={note.id}>{note.note}</NoteBox>
);
});
return (
<section id="cd-timeline">
{noteNodes}
</section>
);
}
});
var NoteBox = React.createClass({
render: function() {
return (
<div className="cd-timeline-block">
<div className="cd-timeline-img">
```

```
<img src={this.props.avatarUri} className="img-circle" alt="Leanna!" />
</div>
<div className="cd-timeline-content">
<h2><a href="#">{this.props.username}</a></h2>
<p>{this.props.children}</p>
<span className="cd-date">{this.props.date}</span>
</div>
</div>
);
}
});
window.NoteSection = NoteSection;
```

The ReactJS App

This is a small ReactJS app that uses our API to build all of the same markup that we had on the page before, but dynamically. It uses jQuery to make the AJAX call:

69 lines web/js/notes.react.js

```
var NoteSection = React.createClass({
... lines 2 - 12
loadNotesFromServer: function() {
$.ajax({
url: '/genus/octopus/notes',
success: function (data) {
this.setState({notes: data.notes});
}.bind(this)
});
},
... lines 21 - 32
});
... lines 34 - 69
```

But I have a hardcoded URL right now - /genus/octopus/notes. Obviously, that's a problem, and lame. But ignore it for a second.

Back in the template, we need to start up the ReactJS app. Add a script tag with type="text/babel" - that's a React thing. To boot the app, add ReactDOM.render:

38 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 23
{% block javascripts %}
... lines 25 - 29
<script type="text/babel" src="{{ asset('js/notes.react.js') }}"></script>
<script type="text/babel">
ReactDOM.render(
... lines 33 - 34
);
</script>
{% endblock %}
```

PhpStorm is *not* going to like how this looks, but ignore it. Render the NoteSection into document.getElementById('js-notes-wrapper'):

38 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 31
ReactDOM.render(
<NoteSection />,
document.getElementById('js-notes-wrapper')
);
... lines 36 - 38
```

Back in the HTML area, clear things out and add an empty div with this id:

38 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 20
<div id="js-notes-wrapper"></div>
{% endblock %}
... lines 23 - 38
```

Everything will be rendered here.

Ya know what? I think we should try it. Refresh. It's alive! It happened quickly, but this *is* loading dynamically. In fact, I added some simple magic so that it checks for new comments every two seconds. Let's see if it'll update without refreshing.

In the controller, remove one of the notes - take out AquaWeaver in the middle. Back to the browser! Boom! It's gone. Now put it back. There it is! So, really cool stuff.

Generating the URL for JavaScript

But... we still have that hardcoded URL. That's still lame, and a problem. *How* you fix this will depend on if you're using AngularJS, ReactJS or something else. But the idea is the same: we need to pass the dynamic value into JavaScript. Change the URL to this.props.url:

69 lines web/js/notes.react.js

```
var NoteSection = React.createClass({
... lines 2 - 12
loadNotesFromServer: function() {
$.ajax({
url: this.props.url,
... lines 16 - 18
});
},
... lines 21 - 32
});
... lines 34 - 69
```

This means that we will pass a url property to NoteSection. Since we create that in the Twig template, we'll pass it in there.

First, we need to get the URL to the API endpoint. Add var notesUrl = ''. Inside, generate the URL with twig using path(). Pass it genus_show_notes and the genusName set to name:

40 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 23
{% block javascripts %}
... lines 25 - 30
<script type="text/babel">
var notesUrl = '{{ path('genus_show_notes', {'genusName': name}) }}';
... lines 33 - 37
</script>
{% endblock %}
```

Yes, this is Twig inside of JavaScript. And yes, I know it can feel a little crazy.

Finally, pass this into React as a prop using url={notesUrl}:

40 lines app/Resources/views/genus/show.html.twig

```
... lines 1 - 33
ReactDOM.render(
<NoteSection url={notesUrl} />,
document.getElementById('js-notes-wrapper')
);
... lines 38 - 40
```

Try that out. It still works very nicely.

Go Deeper!

There is also an open-source bundle called FOSJsRoutingBundle that allows you to generate URLs purely from JavaScript. It's pretty awesome.

Congrats on making it this far: it means you're serious! We've just started, but we've already created a rich HTML page *and* an API endpoint to fuel some sweet JavaScript. And we're just starting to scratch the surface of Symfony.

What about talking to a database, using forms, setting up security or handling API input and validation? How and why should you register your own services? And what are event listeners? The answers to these will make you *truly* dangerous not just in Symfony, but as a programmer in general.

See you on the next challenge.