Starting in Symfony2: Course 1 (2.4+)

With <3 from SymfonyCasts

Chapter 1: Welcome to Symfony!

Go Deeper!

This tutorial has been upgraded! Check out the Symfony 3 Tutorial.

PHP has been on a wild ride over the past few years, with new things like namespaces, our package manager called Composer, and whole lot more sharing and friendship between projects. And in a lot of ways, Symfony has been in the driver's seat for all of that. So congrats on making it here: you're on a path to building great applications.

Let's look at some facts:

Symfony is a component library

This means that you can use any part of it in *any* PHP project, even that ugly legacy app they make you work. We could use Symfony's Router component to give it flexible URLs. Oh, and a lot of other projects use parts of Symfony, like Drupal, Laravel

Symfony is also a "framework"

And because we don't have time to worry about putting a bunch of libraries together, we have the Symfony framework. This is a "suggestion" of how everything can work together in a consistent system. We'll be using the Symfony framework in this course.

Silex is another framework built on Symfony and its goal is to be tiny & really easy to use.

I know what you're wondering: should I use the Symfony framework or this Silex guy? Use them both. Because they both use the same pieces, when you're done with this course, you'll be a certifiable Silex expert.

Symfony plays nice with others

Yep, Symfony tries to be as boring as possible by following widely-used standards. This makes it effortless to include and use other libraries in your project. Want to use a component of Zend Framework? We can do that with just a little bit of configuration.

Symfony is one of the top 3 most watched PHP projects on GitHub

Clearly, you won't be lonely! There's a lot of us out there working with it.

Build Something

In this course, we're going to kick off your Symfony journey by building an events application. To sweeten the deal, you'll also be mastering the most important concepts used across all frameworks and web apps in general, because Symfony steals from, ahem, builds off of the shoulders of other giants.

So even if you don't want to, when you're building something in Symfony, you'll be learning best-practices and probably becoming a better developer. You've been warned!

In this course, we've got some big goals:

1. Start a new project using Symfony2.
2. Explore the core areas, like routing, controllers, requests, responses and templating.
3. Get comfortable with a few extra tools, like Doctrine for dealing with the database and code generation tools.
4. And the real reason you are here, to learn enough best practices and tips to impress your friends at a party.

Ready? Let's go!

Chapter 2: Downloading & Configuration

Go Deeper!

This tutorial has been upgraded! Check out the Symfony 3 Installation.

Ok, let's get Symfony downloaded and setup. Head over to symfony.com and click "Download Now". If the site looks a little different for you, that's because the Internet loves to change things after we record screencasts! But no worries, just find the download page - all the steps will be the same.

Downloading Composer

Tip

Not familiar with Composer? Get spun up by going through the free The Wonderful World of Composer tutorial.

We're going to use a tool called Composer to get the project started. Composer is PHP's package manager. That's a way of saying that it downloads external libraries into our project. Oh yea, and it's also the most important PHP innovation in years.

To get it, go to - you guessed it - GetComposer.org and click Download. Depending on whether you have curl installed, copy one of the two install lines.

Tip

If you're using Windows, someone made an Installer for you :).

Open up your terminal and start typing wildly:

```
$ sdfSFFOOLOOBOO
```

Hmm, ok, that didn't work. So let's try pasting the command instead. If you have PHP 5.4 installed, run this anywhere: we'll use PHP's built-in web server. If you don't, get with it! PHP 5.3 is ancient. But anyways, make sure you have Apache setup and run the command at the server's document root:

```
$ curl -s https://getcomposer.org/installer | php
```

This downloads an executable PHP file called composer.phar. If Composer complains with any warnings or errors, follow its recommendations to make sure your system doesn't panic when we use it.

Downloading the Standard Distribution

Go back to the Symfony.com download page and copy the create-project Composer command. Change the target directory to say starwarsevents and the last part to say @stable. This is the version number and @stable is a neat way of making sure we get the latest and greatest.

```
$ php composer.phar create-project symfony/framework-standard-edition starwarsevent @stable
```

This tells Composer to download the symfony/framework-standard-edition package into the starwarsevents directory. That's all you need to know for now - we're going to explore Composer later.

Ok, so downloading everything is going to take a few minutes. Composer is busy with 2 things at once. First, it's downloading a little example project that uses the Symfony libraries. Even while Composer is doing its thing, we can open a new terminal and move into the new directory. It contains a relatively small number of files and directories:

```
$ cd starwarsevent
$ ls -l
```

Second, our project depends on a bunch of 3rd-party libraries and Composer is downloading these into the vendor/ directory. If you run ls vendor/, you'll see that more and more things are popping up here.

When that finishes, our terminal will become self-aware and start asking us configuration questions. Just hit enter through all of these - we can tweak the config later.

Setup Checks and Web Server Config

Let's make sure that our computer is ready to run Symfony. The project has a little PHP file - web/config.php - that checks our computer and tells us if we're super-heroes on system setup or if our machine is missing some libraries.

We need to navigate to this script in our browser. So if you have PHP 5.4 or higher, just use the built-in PHP web server. Run this command from the root of our project:

```
$ php app/console server:run
```

If you get an error or are using Apache, we have a note on this chapter's page about all that.

Tip

This is just a shortcut for:

```
$ cd web/
$ php -S localhost:8000
```

We now have a web server running at http://localhost:8000, which uses the web/ directory as its doc root. We can just surf directly to the config.php file:

```
http://localhost:8000/config.php
```

Tip

If you're using Apache instead and downloaded the project to your Apache document root, then you can go to "localhost" and find your way to the config.php script:

```
http://localhost/starwarsevents/web/config.php
```

We'll talk more about a proper web server setup later.

If you see any scary "Major Problems", you'll need to fix those. But feel free to ignore any "Minor Problems" for now.

Permissions Craziness

You may see two major issues - permissions problems with the cache and logs directories. Ok, since this can be *really* annoying, we gotta get it fixed.

Basically, we need the cache and logs directories to be writable by our terminal user *and* our web server's user, like www-data. And if a cache file is created by one user, that file needs to be modifiable by the other user. It's an epic battle of 2 UNIX users needing to mess with the same set of files.

Tip

If you're screaming , "If Symfony just creates cache files with 777 permissions, this wouldn't be an issue!", you're right! But that would be a security no-no for shared hosting #sadpanda

Of course, you're awesome and are using the PHP built-in web server. For us, our terminal user *is* our PHP web server user, so we don't have any issues.

If you're using Apache or *are* having issues, check out the sidebar on this page with some tips.

Tip

Fixing Permissions Issues

The easiest permissions fix is to add a little umask() function to the top of 2 files. Pop open your project in your favorite editor, we *love* PhpStorm.

Open up app/console and web/app_dev.php. You'll see a little umask line there - uncomment this:

```
#!/usr/bin/env php
<?php

umask(0000);
// ...
```

> What the heck? The umask function makes it so that cache and logs files are created as 777 (world writable).

Once you're done, set the permissions on the two cache and logs directories:

```
$ chmod -R 777 app/cache/* app/logs/*
```

You shouldn't have any more issues, but if you do, just set the permissions again.

This method *can* be a security issue if you're deploying to a shared server. Check out Symfony's installation chapter for details on other ways to setup your permissions.

Loading up the First Page

Ok, we're ready to get to work. Check out our first real Symfony page, by hitting the app_dev.php file in your browser:

http://localhost:8000/app_dev.php

Hopefully a cute welcome page greets you. The project came with a few demo pages and you're looking at the first one. The code for these lives in the src/Acme/DemoBundle directory. You can see the rest of the demo pages by clicking the "Run The Demo" button.

Tip

If you're using Apache with the same setup as we've done, then the URL will be:

```
http://localhost/starwarsevents/web/app_dev.php
```

Directory Structure

Without writing any code, we already have a working project. Yea, I know, it's kinda lame and boring now, but it *does* have the normal directory structure.

app

Let's look at the app/ dir. It holds configuration and a few other things that tie the whole project together. If your app were a computer, this would be the motherboard: it doesn't really do anything, but it controls everything.

Most of our code will live somewhere else, in directories called "bundles". These bundles are activated in the AppKernel class and configured in the config.yml file inside app/config/.

For example, there's a core bundle called FrameworkBundle. It controls a lot of things, including the session timeout length. So if we needed to tweak this, we'd do it under the framework config key:

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

framework:
    # ...
    session:
        cookie_lifetime: 1440
```

Routes are the URLs of your app, and they also live in this directory in the routing.yml file. We'll master routes in a few minutes.

You can ignore everything else in the app/config/ directory - we'll talk more about them when we cover environments.

The app/ directory is also where the base layout file (app/Resources/views/base.html.twig) and console script (app/console) live. More on those soon!

bin

After app/, we have bin/. You know what? Just forget you ever saw this directory. It has some executable files that Composer added, but nothing we'll ever need at this point.

Tip

Curious about the secrets behind Composer and this bin/ directory. Then do some homework!

src

All the magic and code-writing happens in the src/ directory. We're going to fill it with sub-directories called "bundles". The idea is that each bundle has the code for a single feature or part of your app.

We're about 10 seconds away from nuking it, but if you want to enjoy the demo code, it lives here inside AcmeDemoBundle.

vendor

We already know about the vendor/ directory - this is where Composer downloads outside libraries. It's kinda fat, with a ton of files in it. But no worries, you don't need to look in here, unless you want to dig around in some core files to see how things work. Actually, I love doing that! We'll tear open some core files later.

web

The last directory is web/. It's simple: this is your document root, so put your public stuff here, like CSS and JS files.

There are also two PHP files here that actually execute Symfony. One loads the app in the dev environment (app_dev.php) and the other in the prod environment (app.php). More on this environment stuff later.

Removing Demo Code

It's time to get serious, so let's get all of that demo code out of the way. First, take your wrecking ball to the src/Acme directory:

```
$ rm -rf src/Acme
```

Next, take out the reference to the bundle in your AppKernel so Symfony doesn't look for it when it's loading:

```
// app/AppKernel.php
// ...

if (in_array($this->getEnvironment(), array('dev', 'test'))) {
    // delete the following line
    $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
    $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
    $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
    $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
}
```

Finally, get rid of the _acme_demo route import in the routing_dev.yml file to fully disconnect the demo bundle:

```
# app/config/routing_dev.yml
# ...

# Please! Delete me (the next 2 lines!)
_acme_demo:
    resource: "@AcmeDemoBundle/Resources/config/routing.yml"
```

Refresh your browser. Yes, an error! No, I'm serious, this is good - it's telling us that the page can't be found. The demo page that was here a second ago is gone. Congratulations on your completely fresh Symfony project.

Setting up Git

Let's make our first commit! We're going to use git but not much is different if you use something else. If you don't use version control, shame!

If you already have a .git directory, get rid of it! Otherwise, you'll inherit the history from Symfony's standard distribution, which is about 1000 commits.

```
$ rm -rf .git
```

Create a new repository with git init:

```
$ git init
```

Now don't go crazy with adding files: there are some things that we don't want to commit. Fortunately, Symfony gives us a solid .gitignore file to start with.

The bootstrap.php.cache file is generated when you run Composer. It's super important, though you'll never need to look at it. Regardless, since it's generated automatically, we don't need to commit it.

The cache and logs directories also have generated contents, so we should ignore those too.

The app/config/parameters.yml file holds all server-specific config, like your database username and password. By ignoring it, each developer can keep their own version of the file.

To make life easier, we *do* commit an example version of the file called parameters.yml.dist. That way, a new dev can actually create their parameters.yml file, without guessing what it needs to look like.

We also ignore the vendor/ directory, because Composer downloads everything in here for us. If a new dev clones the code, they can just run php composer.phar install and **bam**, their vendor/ directory looks just like yours.

Everything is being ignored nicely so let's go crazy and add everything to git and commit:

```
$ git add .
$ git commit -m "It's a celebration!!!!!!!"
```

Tip

Unless you want to accidentally commit vacation photos and random notes files, don't run try to avoid running git add ., or at least run git status before committing.

Find some friends! It's time to celebrate the first to your awesome project. Do some jumping high fives, grab a pint, and make a Chewbacca cry.

Chapter 3: Bundles of Joy!

Bundles are a hipster buzzword in the Symfony world. Yea, they're cool but we really deserve all the credit! A bundle is just a place for us to put our hard-earned code. We might make an EventBundle directory for that feature and a UserBundle where we build the registration and login stuff.

We'll put anything and everything into a bundle: PHP code, config, templates, CSS and cats. We can also put other people's bundles into our project. A bundle in Symfony is similar to a plugin in other systems, but, ya know, way more hipster.

The Console

Yes, we *can* create bundles manually. But I'd rather have someone else do it for me. Meet console: a magic executable file in the app/. Run it to see all of the tricks it knows:

```
$ php app/console
```

Woh! All those green words are different console commands, including a lot things that help you work with the database and debug. I like tools as much as any programmer geek, so we'll use a lot of these over time.

Generating the EventBundle

For now run the generate:bundle command:

```
$ php app/console generate:bundle
```

For the bundle namespace, type Yoda/EventBundle. A bundle namespace always has two parts: a vendor name and a name describing the bundle. In honor of the Jedi master, we'll use "Yoda" for the first part and EventBundle for the second. Unless you also work for Yoda, you'll probably use your company or project name instead. Keep these as short as possible to save typing later.

Next, it wants a nickname for our bundle. We're going to be writing this a lot, and lets face it, we're busy people. So, let's choose something short, like EventBundle. The only rule is that this ends with Bundle.

Use the target default directory, but choose yml as the configuration format. You'll just have to trust me on this part - we'll check out the annotation configuration format later.

For the rest of the questions, just hit the enter key wildly. And once the console-gnomes are finished, we have a brand new bundle.

What the Generator Did

This did exactly three things for us.

First, it made a src/Yoda/EventBundle directory with some sample bundle files.

Second, it plugged our bundle into the motherboard by adding a line in the AppKernel class.

Third, it added a line to the routing.yml file that imports routes from the bundle. Contain your excitement: we're about 30 seconds from talking about this part.

The PHPStorm Symfony Plugin

But I want to share a quick secret first. If you're using PHPStorm like I am, I need you to download an aewsome Symfony plugin. For everyone else, this is totally *not* needed, it just adds some shortcuts.

Once it's installed, you need to activate it. We're now super-charged with a ton of Symfony-specific help. You'll see this along the way.

Chapter 4: Routing: The URLs of the World

Go Deeper!

This tutorial has been upgraded! Check out the Symfony 3 Routing.

Let's face it, every page needs a URL. When you need a new page, we always start by creating a route: a chunk of config that gives that page a URL. In Symfony, all routes are configured in just one file: app/config/routing.yml.

Head back to your browser and put /hello/skywalker after app_dev.php:

> http://localhost:8000/app_dev.php/hello/skywalker

The code behind this impressive page was generated automatically in the new bundle. You can change the last part of the URL to anything you want and it greets you politely.

The fact that this page works means that there's a route somewhere that defines this URL pattern. I already said that all routes live in routing.yml, so it *should* be there.

Route Importing

Surprise! It's not here. But there *is* an event entry that was added when we generated the bundle:

```
# app/config/routing.yml
event:
    resource: "@EventBundle/Resources/config/routing.yml"
    prefix:   /
```

The resource key works like a PHP include: point it at another routing file Symfony will pull it in. So, even though Symfony only reads this one routing file, we can pull in routes from anywhere.

Tip

With a little extra work, you could even do cool stuff like loading routes from a custom database table.

Tip

The event key has no significance when importing other routing files.

So what's up with the @EventBundle magic? The resource should just point to the path of another file, relative to this one. But if the file lives in a bundle directory, we can use @ and then the nickname we gave that bundle. Since EventBundle lives at src/Yoda/EventBundle, that's where we'll find the imported file.

Basic Routing

Ah hah! We found the missing route, which makes the /hello/skywalker page work:

```
# src/Yoda/EventBundle/Resources/config/routing.yml
event_homepage:
    pattern:  /hello/{name}
    defaults: { _controller: EventBundle:Default:index }
```

The pattern is the URL and the {name} of the pattern acts like a wildcard. It means that any URL that looks like /hello/* will match this route. If we change hello to there-is-another, the URL to the page changes:

```
# src/Yoda/EventBundle/Resources/config/routing.yml
event_homepage:
    # you can change the URL (but change it back after trying this!)
    pattern:  /there-is-another/{name}
    defaults: { _controller: EventBundle:Default:index }
```

Update the URL in your browser to see the moved page (and then be cool and change the pattern back to /hello/{name}):

> http://localhost:8000/app_dev.php/there-is-another/skywalker

path versus pattern: no difference

Ok, so when you generate your bundle, your route might have path instead of pattern. Scandal!

Here's the story. Once upon a time, the Symfony elders renamed pattern to path, just because it's more semantically correct. And hey, it's shorter anyways. But pattern still works and will until Symfony 3.0. Sorry, that's about as scandalous as things get around Symfony.

To be with the new, I'll change my routing to use path:

```
# src/Yoda/EventBundle/Resources/config/routing.yml
event_homepage:
    path:  /hello/{name}
    defaults: { _controller: EventBundle:Default:index }
```

Tip

But why was it generated as pattern? When we recorded this, the bundle that does the generation magic hadn't released their fix for this change.

The defaults _controller key is the second critical piece of every route. It tells Symfony which controller to execute when the route is matched. But a controller is just a fancy word for a PHP function. So you write this controller function and Symfony executes it when the route is matched.

The _controller Syntax

I know, the EventBundle:Default:index controller doesn't look like any function name you've ever met.

In reality, it's a top-secret syntax with three different parts:

- the bundle name
- the controller class name
- and the method name.

Symfony maps this to a controller class and method:

```
_controller: **EventBundle**:**Default**:**index**

src/Yoda/**EventBundle**/Controller/**Default**Controller::**index** Action()
```

Stop! Let's stare at this for a few seconds, because we're going to see it a lot.

Notice that Symfony adds the word Controller to the end of the class, and Action to the end of the method name. You'll probably hear the method name referred to as an "action".

Open up the controller class and find the indexAction() method:

```
// src/Yoda/EventBundle/Controller/DefaultController.php
namespace Yoda\EventBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    public function indexAction($name)
    {
        return $this->render(
            'EventBundle:Default:index.html.twig',
            array('name' => $name)
        );
    }
}
```

Routing Parameters and Controller Arguments

First, check out the $name variable that's passed as an argument to the method. This is sweet because the value of this argument comes from the {name} wildcard in our route. So if I go to /hello/edgar, the name variable is edgar. When I go to /hello/skywalker, it's skywalker.

And if we change {name} in the route to something else like {firstName}, we'll see an error:

```
# src/Yoda/EventBundle/Resources/config/routing.yml
event_homepage:
    path:  /hello/{firstName}
    defaults: { _controller: EventBundle:Default:index }
```

```
Controller "Yoda\EventBundle\Controller\DefaultController::indexAction()"
requires that you provide a value for the "$name" argument (because there
is no default value or because there is a non optional argument after
this one).
```

Ah hah! So the name of the argument needs to match the name used in the route. Now, the route still has the same URL, we've just given the routing wildcard a different name internally:

```
// src/Yoda/EventBundle/Controller/DefaultController.php
// ...

public function indexAction($firstName)
{
    return $this->render(
        'EventBundle:Default:index.html.twig',
        array('name' => $firstName)
    );
}
```

Let's get crazy by putting a second wildcard in the route path:

```
# src/Yoda/EventBundle/Resources/config/routing.yml
event_homepage:
    path:  /hello/{firstName}/{count}
    defaults: { _controller: EventBundle:Default:index }
```

When we refresh, we get a "No route found" error. We need to put *something* for the count wildcard, other wise it won't match our route. Add /5 to the end to see the page:

> http://localhost:8000/app_dev.php/hello/skywalker/5

Now that we have a count wildcard in the route, we can of course add a $count argument to the action:

```
// src/Yoda/EventBundle/Controller/DefaultController.php

// ...
public function indexAction($firstName, $count)
{
    var_dump($firstName, $count);die;
    // ...
}
```

To prove everything's working, let's dump both arguments. One neat thing is that the order of the arguments doesn't matter. To prove it, swap the order of the arguments and refresh:

```
// src/Yoda/EventBundle/Controller/DefaultController.php

// ...
public function indexAction($count, $name)
{
    // still prints "skywalker" and then "5"
    var_dump($name, $count);die;
    // ...
}
```

We've seen this twice now: Symfony matches the routing wildcards to method arguments by matching their names.

Remove the var_dump code so our page works again.

Routing is full of lots of cool tricks and we'll discover them along the way.

Debugging Routes

Wondering what other URLs your app might have? Our friend console can help you with that with the router:debug command:

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

This shows a full list of every route in your app. Right now, that means the one we've been playing with plus a few other internal Symfony debugging routes. Remember this command: it's your Swiss army knife for finding your way through a project.

Chapter 5: Controllers: Get to work!

Go Deeper!

This tutorial has been upgraded! Check out the [Symfony 3 Controllers][Symfony 3 Controllers].

3 steps. That's all that's behind rendering a page:

1. The URL is compared against the routes until one matches.
2. Symfony reads the _controller key and executes that function.
3. We build the page inside the function.

The controller is all about us, it's where we shine. Whether the page is HTML, JSON or a redirect, we make that happen in this function. We might also query the database, send an email or process a form submission here.

Tip

Some people use the word "controller" to both refer to a the class (like DefaultController) *or* the action inside that class.

Returning a Response

Controller functions are dead-simple, and there's just one big rule: it must return a Symfony :symfonyclass:Symfony\\Component\\HttpFoundation\\Response object.

To create a new Response, add its namespace to top of the controller class. I know, the namespace is horribly long, so this is where having a smart IDE like PHPStorm will make you smile:

```
// src/Yoda/EventBundle/Controller/DefaultController.php
namespace Yoda\EventBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

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

Tip

If you're new to PHP 5.3 namespaces, check out our [free screencast on the topic][free screencast on the topic].

Now create the new Response object and quote Admiral Ackbar:

```
public function indexAction($count, $firstName)
{
    return new Response('It\'s a traaaaaaaap!');
}
```

Now, our page has the text and nothing else.

Again, controllers are simple. No matter how complex things seem, the goal is always the same: generate your content, put it into a Response, and return it.

Returning a JSON Response

How would we return a JSON response? Let's create an array that includes the $firstName and $count variables and turn it into a string with json_encode. Now, it's exactly the same as before: pass that to a Response object and return it:

```
public function indexAction($count, $firstName)
{
    $arr = array(
        'firstName' => $firstName,
        'count'     => $count,
        'status'    => 'It\'s a traaaaaaaap!',
    );

    return new Response(json_encode($arr));
}
```

Now our browser displays the JSON string.

Tip

There is also a [JsonResponse][JsonResponse] object that makes this even easier.

Wait. There *is* one problem. By using my browser's developer tools, I can see that the app is telling my browser that the response has a text/html content type.

That's ok - we can fix it easily. Just set the Content-Type header on the Response object to application/json:

```
public function indexAction($count, $firstName)
{
    // ...

    $response = new Response(json_encode($arr));
    $response->headers->set('Content-Type', 'application/json');

    return $response;
}
```

Now when I refresh, the response has the right content-type header.

I know I'm repeating myself, but this is important I promise! Every controller returns a Response object and you have full control over each part of it.

Rendering a Template

Time to celebrate: you've just learned the core of Symfony. Seriously, by understanding the routing-controller-Response flow, we could do anything.

But as much as I love printing Admiral Ackbar quotes, life isn't always this simple. Unless we're making an API, we usually build HTML pages. We could put the HTML right in the controller, but that would be a Trap!

Instead, Symfony offers you an optional tool that renders template files.

Before that, we should take on another buzzword: services. These are even trendier than bundles!

Symfony Services

Symfony is basically a wrapper around a big bag of objects that do helpful things. These objects are called "services": a techy name for an object that performs a task. Seriously: when you hear service, just think "PHP object".

Symfony has a ton of these services - one sends emails, another queries the database and others translate text and tie your shoelaces. Symfony puts the services into a big bag, called the "mystical service container". Ok, I added the word mystical: it's just a PHP object and if you have access to it, you can fetch any service and start using it.

And here's the dirty secret: everything that you think "Symfony" does, is actually done by some service that lives in the container. You can even tweak or replace core services, like the router. That's really powerful.

In any controller, this is great news because, surprise, we have access to the mystical container via $this->container:

```
public function indexAction($count, $firstName)
{
    // not doing anything yet...
    $this->container;

    // ...
}
```

Tip

This only works because we're in a controller *and* because we're exending the base :symfonyclass:Symfony\\Bundle\\FrameworkBundle\\Controller\\Controller class.

One of the services in the container is called templating. I'll show you how I knew that in a bit:

```
public function indexAction($count, $firstName)
{
    $templating = $this->container->get('templating');

    // ...
}
```

This templating object has a render method on it. The first argument is the name of the template file to use and the second argument holds the variables we want to pass to the template:

```
// src/Yoda/EventBundle/Controller/DefaultController.php
// ...

public function indexAction($count, $firstName)
{
    $templating = $this->container->get('templating');

    $content = $templating->render(
        'EventBundle:Default:index.html.twig',
        array('name' => $firstName)
    );

    // ...
}
```

The template name looks funny because it's another top secret syntax with three parts:

- the **bundle name**
- a **directory name**
- and the **template's filename**.

```
EventBundle:Default:index.html.twig

src/Yoda/EventBundle/Resources/views/Default/index.html.twig
``[

This looks like the ][

This looks like the ]controller` syntax we saw in routes, but don't mix
them up. Seriously, one points to a controller class & method. This one points
to a template file.

Open up the template.
```

{# src/Yoda/EventBundle/Resources/views/index.html.twig #}

Hello {{ name }}

```


Welcome to Twig! A curly-little templating language that you're going to
fall in love with. Right now, just get fancy by adding a strong tag:
```

Hello **{{ name }}**

```
Back in the controller, the `render` method returns a string. So just like
before, we need to put that into a new `Response` object and return it:
```

public function indexAction($count, $firstName) {

```
$templating = $this->container->get('templating');

$content = $templating->render(
    'EventBundle:Default:index.html.twig',
    array('name' => $firstName)
);

return new Response($content);
```

}

```
Refresh. There's our rendered template. We still don't have a fancy layout,
just relax - I can only go so fast!

## Make this Shorter

Since rendering a template is pretty darn common, we can use some shortcuts.
First, the `templating` service has a `renderResponse` method. Instead
of returning a string, it puts it into a new `Response` object for us.
Now we can remove the `new Response` line and its `use` statement:
```

// src/Yoda/EventBundle/Controller/DefaultController.php namespace Yoda\EventBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller {

```
public function indexAction($count, $firstName)
{
    $templating = $this->container->get('templating');

    return $templating->renderResponse(
        'EventBundle:Default:index.html.twig',
        array('name' => $firstName)
    );
}
```

}

```
### And even Shorter

Better. Now let's do less. Our controller class extends Symfony's own base
controller. That's optional, but it gives us shortcuts.

[Open up the base class][Open up the base class], I'm using a "go to file" shortcut in my editor to
search for the `Controller.php` file.

One of its shortcut is the `render` method. Wait, this does exactly what
we're already doing! It grabs the `templating` service and calls `renderResponse`
on it:
```

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

public function render($view, array $parameters = array(), Response $response = null) {

```
return $this->container->get('templating')->renderResponse(
    $view,
    $parameters,
    $response
);
```

}

```
Let's just kick back, call this method and return the result:
```

public function indexAction($count, $firstName) {

```
return $this->render(
    'EventBundle:Default:index.html.twig',
    array('name' => $firstName)
);
```

}

```
I'm sorry I made you go the long route, but now you know about the container
and how services are working behind the scenes. And as you use more shortcut
methods in Symfony's base controller, I'd be so proud if you looked to see
what each method *actually* does.

Controllers are easy: put some code here and return a `Response` object.
And since we have the container object, you've got access to every service
in your app.

Oh right, I haven't told you what services there are! For this, go back to
our friend console and run the `container:debug` command:
```

php app/console container:debug

```
It lists every single service available, as well as what type of object it
returns. Color you dangerous.

Ok, onto the curly world of Twig!


[free screencast on the topic]: http://knpuniversity.com/screencast/php-namespaces-in-120-seconds
[JsonResponse]: http://symfony.com/doc/current/components/http_foundation/introduction.html#creating-a-json-response
[Open up the base class]:
https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php
[Symfony 3 Controllers]: http://knpuniversity.com/screencast/symfony/first-page
```

Chapter 6: Twig

Now for the absolutely hardest part of Symfony. I'm kidding! We're talking about templates: those files where we mix HTML tags and dynamic data to build our page. We already saw that these are rendered using the templating service, usually from inside a controller.

We also know that they're written in Twig, a language that feels a lot like PHP, but was made specifically to be awesome at doing templating tasks, like looping, rendering other templates, and handling layouts.

If you haven't seen Twig before, trust me, you're going to love it! It's easy and a joy to work with.

Tip

If you *do* end up hating Twig, you can use normal PHP templates as well, though 3rd party bundles don't support these as well.

The Print Syntax

Twig only has two different tags and we've already seen the first one: the print tag:

```
{{ name }}
```

No matter what, if you want to print something in Twig, then you'll use this double curly brace format. I call this the "say something" tag, and it's basically the same as opening PHP and using echo:

```
<?php echo $name ?>
```

In our template, we're printing the name variable, which was passed to the template from the controller.

The Do Something Sytnax

The second tag is the "do something" tag. Its syntax is {% %} and we use it to do things like looping, defining variables, and if statements. It's easier to see, so let's pass the count variable into our template:

```
// src/Yoda/EventBundle/Controller/DefaultController.php
// ...

public function indexAction($count, $firstName)
{
    return $this->render(
        'EventBundle:Default:index.html.twig',
        array('name' => $firstName, 'count' => $count)
    );
}
```

Now, use Twig's for tag to print the name a certain number of times:

```
{# src/Yoda/EventBundle/Resources/views/Default/index.html.twig #}

{% for i in 1..count %}
    Hello <strong>{{ name }}</strong> # {{ i }}!<br/>
{% endfor %}
```

Now refresh! There are a finite number of "do something" tags and there's even a handy list on Twig's Documentation page. Scroll down a little bit and check out the list on the left.

Tip

Symfony adds just a few more Twig "do something" tags. Find them in the Symfony Reference documentation.

So the {{ }} syntax prints things and the {% %} performs other language actions. If you've got this down, I'd say you've just about mastered Twig.

Comments

Ok, I hate to lie to you, so there *is* a third tag, but it's just used to write comments.

```
{# Hello comments! #}
```

Filters

But wait, there's more! Twig has a lot of nice tricks and sugar, like filters! We can use the upper filter to capitalize the name variable:

```
Hello <strong>{{ name|upper }}</strong> # {{ i }}!<br/>
```

If you're used to piping things together in a UNIX terminal, this works the same way. Back on the Documentation page, you'll find a big list of filters, You can even use filters on top of filters.

Twig also has functions and a cool thing called tests, which lets you write things like {% if i is odd %}. But that's all just extra fun stuff.

Tip

If you want to get deeper with these types of tricks or want to help your frontend designer get started, check out our Twig Screencast.

Extending a Base Layout

Despite all my Twig hype, our template is depressing: it's got some HTML, but no layout. If only we had a base layout template that could decorate all of our page.

Oh right, there *is* one, and it lives in the app/Resources/views directory. Actually, it's kind of plain too, but has a basic HTML structure:

```
{# 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>
```

To use this layout, we "extend" it. First, add the extends tag to the top of the index.html.twig template. Now, wrap everthing else in a {% block body %} tag:

```
{# src/Yoda/EventBundle/Resources/views/Default/index.html.twig #}
{% extends '::base.html.twig' %}

{% block body %}
    {# ... the rest of the template ... #}
{% endblock %}
```

Refresh and check out the source. The HTML from base.html.twig is being used and the content from our template is rendered in the middle of it.

Twig Blocks

Let's break this down. The extends tag says that we want to *dress* our template with another template. Inside base.html.twig, we have a bunch of block tags. One of them is called body and looks just like what we added to *our* template.

```
{# 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>
```

Blocks define "holes" that a child template can fill in. The content in the body block of index.html.twig is inserted into the body block of base.html.twig.

There's also a title block, which already has content in it:

```
<title>{% block title %}Welcome!{% endblock %}</title>
```

This block has default content, which is working since the page's title is indeed Welcome!.

Let's replace it with something a bit less boring. We know how to do this now, just add a title block to index.html.twig.

```
{# src/Yoda/EventBundle/Resources/views/Default/index.html.twig #}
{% extends '::base.html.twig' %}

{% block title %}Some Twiggy Goodness{% endblock %}

{% block body %}
    {# ... #}
{% endblock %}
```

And to be even lazier, there's a shorter syntax for simple blocks like this:

```
{% block title 'Some Twiggy Goodness' %}
```

The blocks ones in base.html.twig are just suggestions, feel free to change their names or add some more to have an even more flexible layout.

Tip

Yes, you can also append to a block instead of replacing it. This is done with parent() and we chat about it in Episode 2.

"::base.html.twig" Naming

The ::base.html.twig filename looks weird. But it's actualy the exact same syntax we're using in our controller, just in disguise!

Remember that a template name always has three parts:

- The bundle name
- A subdirectory
- And the template filename

In this case, the bundle name and subdirectory are just missing. When a template name *has* the bundle part, it means the template lives in the Resources/views directory of that bundle. But when this part is missing, like here, it means the template lives in the *app*/Resources/views directory. And since the second part is missing too, it means it lives directly there, and not in a subdirectory.

Tip

Template name and path examples

- EventBundle:Default:index.html.twig

  src/Yoda/EventBundle/Resources/views/Default/index.html.twig

- EventBundle::index.html.twig

  src/Yoda/EventBundle/Resources/views/index.html.twig

- ::base.html.twig

  app/Resources/views/index.html.twig

Web Debug Toolbar

In the browser, we're now staring at a killer feature of Symfony2: the polite little bar on the bottom. This is the web debug toolbar, and you may end up loving it even more than the console.

It tells us which controller was rendered, the page load time, memory footprint, security info, form details and more. It's added automatically to any page that has a valid HTML structure. That's why we didn't see it until we extended the layout file.

Click anywhere on it to multiply the amount of information it gives you by 100! This is the profiler, which is broken down into sections. The best one is the Timeline. It visually tells us *exactly* what's going on during a request and how much time everything is taking. A lot of what you see here are background Symfony events.

Chapter 7: Databases and Doctrine

Symfony doesn't care about your database or the code you use to talk to it. Seriously. It's not trying to be rude, but other libraries already solve this problem. So if you want to make a :phpclass:PDO connection and run raw SQL queries, that's great! When we create services in [Episode 3](), you'll learn some life-saving strategies to organize something like this.

But most people that use Symfony use a third-party library called Doctrine. It has its own [website and documentation](), though [Symfony's Doctrine documentation]() is a lot friendlier.

In a nutshell, Doctrine maps rows and columns in your database to objects and properties in PHP. Imagine we have an Event object with name and location properties. If we tell Doctrine to save this object, it inserts a row into a table and puts the data on name and location columns. And when we query for the event, it puts the column data back onto the properties of an Event object.

The big confusing mind-switch is to stop thinking about tables and start thinking about PHP classes.

Creating the Event Entity Class

In fact, let's create the Event class we were talking about. The console can even make this for us with the doctrine:generate:entity command:

```
$ php app/console doctrine:generate:entity
```

Like other commands, this one is self-aware and will start asking you questions. In step 1, enter EventBundle:Event. This is another top-secret shortcut name and it means you want the Event class to live inside the EventBundle.

Now, choose annotation as the configuration format and move on to field creation. Add the following fields:

- name as a string field;
- time as a datetime field;
- location as a string field;
- and details as a text field.

These types here are configuration that tell Doctrine how each property should be stored in the database.

If you messed anything up, panic! Or just exit with ctrl+c try the command again. Nothing happens until it finishes.

Tip

All of the Doctrine data types are explained in their documentation: [Doctrine Mapping Types]().

Say "yes" for the repository class and confirm generation. A repository is a cool guy we'll use later to store custom queries.

What just Happened?

Ok! So what did that do? Actually, it just created 2 new classes in an Entity directory in our bundle. And that's it.

Check out the new Event class:

```
// src/Yoda/EventBundle/Entity/Event.php
namespace Yoda\EventBundle\Entity;

/**
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Yoda\EventBundle\Entity\EventRepository")
 */
class Event
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

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

    // ...

    public function getName()
    {
        return $this->name;
    }

    public function setName($name)
    {
        $this->name = $name;
    }

    // ...
}
```

For Doctrine, the word "entity" means a normal PHP class that we will save to the database. So, whenever I say "entity", just scream: "that's just a normal PHP class!". Your co-workers will love you!

If you ignore the PHP comments, you'll see that this is a plain old PHP class. It doesn't do anything: it just stores data on its private properties. Getter and setter methods - like getName() and setName() - were generated so we can play with an event's data. It's underwhelming, almost disappointing, and that's what makes Doctrine so interesting.

Now, check out the PHP comments above the class. These comments are called "annotations", and they're actually read and parsed by Doctrine. So when you hear "annotations", shout "PHP comments that are read like configuration!".

These tell Doctrine *how* it should save an Event object to the database. Right now, they will save to an event table and each property will be a column in that table. I usually like to prefix all of my table names, so let's do that by adding a name option to the Table annotation:

```
/**
 * @ORM\Table(name="yoda_event")
 * @ORM\Entity(repositoryClass="Yoda\EventBundle\Entity\EventRepository")
 */
class Event
{
    // ...
}
```

Creating the "play" Script

We're ready to insert data, but first I want to show you a debugging trick. First, copy the web/app_dev.php file to the root of the project and rename it to play.php:

```
$ cp web/app_dev.php play.php
```

Open it up and remove the IP protection stuff at the top and update the require paths since we moved things around:

```
// play.php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Debug\Debug;
umask(0000);

$loader = require_once __DIR__.'/app/bootstrap.php.cache';
Debug::enable();

require_once __DIR__.'/app/AppKernel.php';
// ...
```

This script boots Symfony, processes the request, and spits out the page. But I have evil plans to transform it into a debugging monster where we can write random code and execute it from the command line to see what happens.

Replace the last three lines with $kernel->boot():

```
// ...
require_once __DIR__.'/app/AppKernel.php';

$kernel = new AppKernel('dev', true);
$kernel->loadClassCache();
$request = Request::createFromGlobals();
$kernel->boot();
```

Remember the service container from earlier? We have access to it here. To make it as flexible as possible, I'll add a few lines that help fake a real request. This is a little jedi mind trick so don't worry about what these do right now:

```
// play.php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Debug\Debug;
umask(0000);

$loader = require_once __DIR__.'/app/bootstrap.php.cache';
Debug::enable();

require_once __DIR__.'/app/AppKernel.php';

$kernel = new AppKernel('dev', true);
$kernel->loadClassCache();
$request = Request::createFromGlobals();
$kernel->boot();

$container = $kernel->getContainer();
$container->enterScope('request');
$container->set('request', $request);

// all our setup is done!!!!!!
```

Our evil creation is alive! So let's play around. How could we render a template here? Why, just by grabbing the templating service and using its render() method:

```
// ...
// all our setup is done!!!!!!
$templating = $container->get('templating');

echo $templating->render(
    'EventBundle:Default:index.html.twig',
    array(
        'name' => 'Yoda',
        'count' => 5,
    )
);
```

Execute the play script from the command line.

```
$ php play.php
```

When I run it, the template is rendered and printed out. How cool is that? This is perfect for whenever we need to quickly test out some code.

Chapter 8: Inserting and Querying Data

We can use the play script to create and save our first Event object. Start by importing the Event class's namespace:

```
// ...
// all our setup is done!!!!!!
use Yoda\EventBundle\Entity\Event;
```

Let's flex our mad PHP skills and put some data on each property. Remember, this is just a normal PHP object, it doesn't have any jedi magic and doesn't know anything about a database:

```
use Yoda\EventBundle\Entity\Event;

$event = new Event();
$event->setName('Darth\'s surprise birthday party');
$event->setLocation('Deathstar');
$event->setTime(new \DateTime('tomorrow noon'));
$event->setDetails('Ha! Darth HATES surprises!!!!');
```

Let's save this wild event! To do that, we use a special object called the "entity manager". It's basically the most important object in Doctrine and is in charge of saving objects and fetching them back out. To get the entity manager, first grab a service from the container called doctrine and call getManager on it:

```
$em = $container->get('doctrine')->getManager();
```

Saving is a two-step process: persist() and then flush():

```
$em = $container->get('doctrine')->getManager();
$em->persist($event);
$em->flush();
```

Two steps! Yea, and for an awesome reason. The first tells Doctrine "hey, you should know about this object". But no queries are made yet. When we call flush(), Doctrine actually executes the INSERT query.

The awesome is that if you need to save a bunch of objects at once, you can persist each of them and call flush once. Doctrine will then pack these operations into as few queries as possible.

Now, when we execute our play script, it blows up!

```
PDOException: SQLSTATE[42000][1049] Unknown database 'symfony'
```

Scroll up to see the error message: "Unknown database symfony". Duh! We skipped one important step: setting up the database config.

Configuring the Database

Database config is usually stored in app/config/parameters.yml. Change the database name to "yoda_event". For my super-secure computer, the database user root with no password is perfect:

```
# app/config/parameters.yml
parameters:
    database_driver:   pdo_mysql
    database_host:     127.0.0.1
    database_port:     null
    database_name:     yoda_event
    database_user:     root
    database_password: null
    # ...
```

We can haz Database

Now, I know you're super-smart and capable, but let's be lazy again and use the console to create the database for us with the doctrine:database:create command:

```
$ php app/console doctrine:database:create
```

There's also a command to drop the database. That's great, until you realize that you just ran it on production... and everyone is running around with their hair on fire. Yea, so that's a healthy reminder to not give your production db user access to drop the database.

A Table for Events, Please

We have a database, but no tables. Any ideas who might help us with this? Oh yeah, our friend console! Run the doctrine:schema:create command. This finds all your entities, reads their annotation mapping config, and creates all the tables:

```
$ php app/console doctrine:schema:create
```

Time to try out the play script again:

```
$ php play.php
```

What? No errors! Did it work? Use the doctrine:query:sql command to run a raw query against the database:

```
$ php app/console doctrine:query:sql "SELECT * FROM yoda_event"
```

And voila! There's our event.

Making nullable Fields

Let's get crazy and leave the details field blank:

```
// play.php
// ...
$event->setTime(new \DateTime('tomorrow noon'));
//$event->setDetails('Ha! Darth HATES surprises!!!!');
```

When we run the script, another explosion! Scrolling up, the error straight from MySQL saying that the details column can't be null.

> SQLSTATE[23000]: Integrity constraint violation: 1048 Column 'details' cannot be null

So Doctrine assumes by default that all of your columns should be set to NOT NULL when creating the table. To change this, add a nullable option to the details property inside the entity:

```
// src/Yoda/EventBundle/Entity/Event.php

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

// ...
```

Tip

Doctrine has a killer page that shows all of the annotations and their options. See Annotations Reference.

But before this does anything, the actual column in the database needs to be modified to reflect the change. Hey, console to the rescue! Run the doctrine:schema:update command:

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

This is pretty sweet: it looks at your annotations mapping config, compares it against the current state of the database, and figures out exactly what queries we need to run to update the database structure.

But the command didn't do anything yet. Pass --dump-sql to see the queries it wants to run and --force to actually run them:

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

Run the play script again. Alright, no errors means that the new event is saved without a problem.

Querying for Objects

Putting stuff into the database is nice, but let's learn how to get stuff out. Open up the DefaultController class we've been playing with. First, we need to get the all-important entity manager. That's old news for us. Like before, just get the doctrine service from the container and call getManager on it.

This works, but since we're extending the base controller, we can use its getDoctrine() to get the doctrine service. That'll save us a few keystrokes:

```
// src/Yoda/EventBundle/Controller/DefaultController.php
// ...

public function indexAction($count, $firstName)
{
    // these 2 lines are equivalent
    // $em = $this->container->get('doctrine')->getManager();
    $em = $this->getDoctrine()->getManager();

    // ...
}
```

To query for something, we always first get an entity's repository object:

```
public function indexAction($count, $firstName)
{
    // these 2 lines are equivalent
    // $em = $this->container->get('doctrine')->getManager();
    $em = $this->getDoctrine()->getManager();
    $repo = $em->getRepository('EventBundle:Event');

    // ...
}
```

A repository has just one job: to help query for one type of object, like Event objects. The EventBundle:Event string is the same top-secret shortcut syntax we used when we generated the entity - it's like the entity's nickname.

Tip

If you like typing, you can use the full class name anywhere the entity "alias" is used:

```
$em->getRepository('Yoda\EventBundle\Entity\Event');
```

Use the repository's findOneBy() method to get an Event object by name. There are other shortcut methods too, like findAll(), findBy(), and find():

```
// src/Yoda/EventBundle/Controller/DefaultController.php
// ...

public function indexAction($count, $firstName)
{
    // these 2 lines are equivalent
    // $em = $this->container->get('doctrine')->getManager();
    $em = $this->getDoctrine()->getManager();
    $repo = $em->getRepository('EventBundle:Event');

    $event = $repo->findOneBy(array(
        'name' => 'Darth\'s surprise birthday party',
    ));

    return $this->render(
        'EventBundle:Default:index.html.twig',
        array(
            'name' => $firstName,
            'count' => $count,
            'event'=> $event,
        )
    );
}
```

Tip

In Episode 2, we'll add more methods to the repository and write some custom queries.

Rendering Entities in Twig

Ok - let's pass the Event object into the template as a variable. We can use Twig's render syntax to print out the name and location properties. Internally, Twig is smart enough to call getName() and getLocation(), since the properties are private:

```
{% block body %}
    {# ... #}

    {{ event.name }}<br/>
    {{ event.location }}<br/>

{% endblock %}
```

Refresh the page! I can see our event data, so all the magic Doctrine querying must be working. Actually, check out out the web debug toolbar. The cute box icon jumped from zero to one, which is the number of queries used for the page. When we click the little boxes, we can even see what those queries are and even run EXPLAIN on them.

Good work young jedi! Seriously, you know the basics of Doctrine, and that's not easy. In the next 2 episodes, we'll create custom queries and use cool things like events that let you "hook" into Doctrine as entities are inserted, updated or removed from the database.

Oh, and don't forget Doctrine has its own documentation, though the most helpful pages are the Annotations Reference and Doctrine Mapping Types reference pages. And by the way, when you see annotations in the Doctrine docs, prefix them with @ORM\ before putting them in Symfony. That's because of this use statement above our entity:

```
// src/Yoda/EventBundle/Entity/Event.php
use Doctrine\ORM\Mapping as ORM;
// ..
```

If that's in your class and you have @ORM\ at the start of all of your Doctrine annotations, you're killing it.

Chapter 9: Virtual Host Setup Extravaganza

We're using the built-in PHP web server, and it's awesome for development. But it only handles one request at a time, so unless you only ever want 1 visitor, we're going to need something different.

To see how this might look, we'll invent a fake domain - events.l - and set it up to point to our project. I'll use Apache, though it's more and more common to use Nginx with PHP-FPM, because they're lightning fast. But all the ideas are the same and we have a page on the official Symfony documentation with some more details.

Creating a VirtualHost

Step 1 is to modify the Apache configuration to create a new VirtualHost. I *would* love to tell you where this lives, but this is one of those files that hides in different places on each system setup. I use Apache via MacPorts, so my virtual hosts live in the /opt directory.

Tip

In Ubuntu, it lives in /etc/apache2/sites-available and each has its own file that needs to be activated. See HTTPD Configuration.

The configuration we need is simple:

```
<VirtualHost *:80>
    ServerName events.l
    DocumentRoot "/Users/leanna/Sites/starwarsevents/web"

    <Directory "/Users/leanna/Sites/starwarsevents/web">
        AllowOverride All
    </Directory>
</VirtualHost>
```

Make sure the DocumentRoot points to the *web* directory of the project so that *only* files inside it are accessible via your browser. Oh, and the AllowOverride All tells Apache that it's ok to use the .htaccess file in the web/ directory.

Tip

For more information, or to see Nginx configuration, see Configuring a Web Server.

Now, restart Apache. Yep, this command *also* varies across systems:

```
$ sudo /opt/local/apache2/bin/apachectl restart
```

Tip

In Ubuntu, the command is:

```
$ sudo service apache2 restart
```

Finally, we need to add a hosts entry to /etc/hosts:

```
# /etc/hosts
# ...

127.0.0.1   events.l
```

This points events.l right back at our local computer. And this file is always at the same location... except for windows.

We have a VirtualHost and the hosts entry, so let's go to http://events.l/app_dev.php. You *may* get a permissions error, and if you do, just chmod 777 your cache and logs directories for now. But longer-term, go back to the installation chapter for details on how to fix this.

The 404 error is fine, because we don't have a homepage yet. Add the path to the page we've been working on after app_dev.php to see it:

> http://events.l/app_dev.php/hello/skywalker/5

The dev and prod Environments

Let's talk more about that app_dev.php script that's always in our URL. A stock Symfony app has two different "modes" called "environments". When you hit app_dev.php, you're running your app in the dev environment. This shows us big descriptive errors, automatically rebuilds the cache, and makes the web debug toolbar popup. It's our debugging hero.

The other environment is prod and it kicks butt by being fast and by turning off debugging tools. To run the app in the prod environment, switch the URL from app_dev.php to app.php:

> http://events.l/app.php/hello/skywalker/5

What!? 404 page! Outrageous!

We can't see the error, but we *can* tail the prod log file:

```
$ tail app/logs/prod.log
```

Hmm, no route found. Ah, of course! Symfony compiles all of its configuration into cache files. So if we change a routing.yml file, the cache needs to be rebuilt. The dev environment does that for us, but for speed reasons, prod doesn't.

To do this, find our friend console and run the cache:clear command with a --env=prod option.

```
$ php app/console cache:clear --env=prod --no-debug
```

The means we're clearing the cache for the prod environment.

Tip

If you haven't properly fixed your permissions <ep1-install-permissions> yet, you'll need to sudo chmod -R 777 app/cache after this command.

Refresh the page to see your functional page in the prod environment:

> http://events.l/hello/skywalker/5

Awesome! But I thought we had put app.php in the URL. Where did it go? Our project came with a web/.htaccess file that have 2 pieces of goodness in it.

First, it has a rewrite rule that sends all requests through app.php, which means we don't need to have it in our URL.

```
# web/.htaccess
# ...

# If the requested filename exists, simply serve it.
# We only want to let Apache serve files and not directories.
RewriteCond %{REQUEST_FILENAME} -f
RewriteRule .? - [L]

# Rewrite all other queries to the front controller.
RewriteRule .? %{ENV:BASE}/app.php [L]
```

Awesome, the app.php was ugly anyways.

Second, even if you *do* put app.php in the URL, it notices that you don't need this and redirects to remove it:

```
# web/.htaccess
# ...

# Redirect to URI without front controller to prevent duplicate content
# (with and without `/app.php`).
RewriteCond %{ENV:REDIRECT_STATUS} ^$
RewriteRule ^app\.php(/(.*)|$) %{ENV:BASE}/$2 [R=301,L]
```

The prod environment is only useful after you deploy. So let's get back to the dev environment so we can see errors.

Chapter 10: Code Generation FTW!

I feel like making someone else do some work for awhile, so let's look at Symfony's code generation tools.

The first thing we need our app to do is let users create, view, update and delete events. In other words, we need a CRUD for the Event entity.

Want to use Doctrine to generate a CRUD? Yea, there's a console command for that doctrine:generate:crud:

```
$ php app/console doctrine:generate:crud
```

This inquisitive command first wants to know which entity we need a CRUD for. Answer with that shortcut entity "alias" name we've been seeing: EventBundle:Event.

Say "yes" to the "write" actions, yml for the configuration format, and use the default /event route prefix. Then finish up.

Routing Imports and Organization

Ah crap! Red errors! It's ok, copy the code into the routing.yml of our EventBundle.

```
# src/Yoda/EventBundle/Resources/config/routing.yml
# ...

# copied in from the commands output
EventBundle_event:
    resource: "@EventBundle/Resources/config/routing/event.yml"
    prefix:   /event
```

The generation tasks tried to put this in there for us, but we already had something in this file so it panicked. All better now.

We now know this is a routing import, which loads a brand new event.yml file:

```
# src/Yoda/EventBundle/Resources/config/routing/event.yml
event:
    pattern:  /
    defaults: { _controller: "EventBundle:Event:index" }

event_show:
    pattern:  /{id}/show
    defaults: { _controller: "EventBundle:Event:show" }

# ... more routes
```

Let's run the router:debug command to make sure these are being loaded:

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

```
event            ANY    /event/
event_show       ANY    /event/{id}/show
event_new        ANY    /event/new
event_create     POST   /event/create
event_edit       ANY    /event/{id}/edit
event_update     POST   /event/{id}/update
event_delete     POST   /event/{id}/delete
```

Check out the main app/config/routing.yml file - it's still only importing the *one* file from the EventBundle:

```
# app/config/routing.yml
event:
    resource: "@EventBundle/Resources/config/routing.yml"
    prefix:   /
```

But once we're in that file, we're of course free to organize routes into even more files and import those. That's what's happening with event.yml: it holds all the routes for the new EventController. Don't go crazy, but when you have a lot of routes, splitting them into multiple files is a good way to keep things sane.

Oh, and when we import another file, the key - like EventBundle_event - is completely meaningless - make it whatever you want. **But**, the key for an actual route *is* important: it becomes its internal "name". We'll use it later when we generate links.

Checking out the Generated Code

Enough with routing! Head to the /event page to see this in action. I know we got Apache setup in the last chapter, but I'm going to continue using the built-in PHP web server and access the site at localhost:8000:

```
http://localhost:8000/app_dev.php/event
```

Woh, that's ugly. Hmm, but it *does* work - we can add, view, update and delete events. Easy!

Let's peek at some of the code. The generated controller is like a cheatsheet for how to do common things, like form processing, deleting entities, redirecting and showing a 404 page.

For example, showAction uses the id from its route to query for an Event object. If one isn't found, it sends the user to a 404 page by calling createNotFoundException and throwing the result. This helper function is just a shortcut to create a very specific type of Exception object that causes a 404 page:

```
// src/Yoda/EventBundle/Controller/Event.php
// ...

public function showAction($id)
{
    $em = $this->getDoctrine()->getManager();

    $entity = $em->getRepository('EventBundle:Event')->find($id);

    if (!$entity) {
        throw $this->createNotFoundException('No event with id '.$id);
    }

    // ...
    return $this->render('EventBundle:Event:show.html.twig', array(
        'entity' => $entity,
        'delete_form' => $deleteForm->createView(),
    ));
}
```

If we do find an Event, it's passed to the template and rendered. Take a few minutes to look through the other parts of the controller. I mean it!

Making the Generated Code Less Ugly

I know this all works, but the ugly is killing me. I created a custom version of each of the CRUD template files while you were looking through the controller. You can find these in the resources directory of the code download for this screencast. I already moved that directory from the code download into my project.

```
$ cp resources/Event/* src/Yoda/EventBundle/Resources/views/Event/
```

The 3-template Inheritance System

If you think that the new template files probably extend a layout file, gold star! But I can't make it that easy. Instead of extending the ::base.html.twig file we're familiar with, each extends EventBundle::layout.html.twig:

```
{# src/Yoda/EventBundle/Resources/views/Event/index.html.twig #}
{% extends 'EventBundle::layout.html.twig' %}

...
```

Let's create this template. The middle piece of the 3-part template syntax is missing, which tells us that this will live directly in the Resources/views directory of our bundle, and not in a sub-directory:

```
{# src/Yoda/EventBundle/Resources/views/layout.html.twig #}

Create this file... but nothing here yet...
```

Inside the new template, simply extend ::base.html.twig:

```
{# src/Yoda/EventBundle/Resources/views/layout.html.twig #}
{% extends '::base.html.twig' %}
```

Now we have a template hierarchy - index.html.twig extends layout.html.twig, which extends base.html.twig.

Tip

If you try the new templates out, and your browser shows the old ones, try clearing out your cache (php app/console cache:clear) - this could be a rare time when Symfony doesn't rebuild the cache correctly.

This is awesome because *all* the new templates extend layout.html.twig. So if we want to override a block for *all* of our event pages, we can do that right here.

Let's try it: set the title block to "Events":

```
{# src/Yoda/EventBundle/Resources/views/layout.html.twig #}
{% extends '::base.html.twig' %}

{% block title 'Events' %}
```

Now we have a better default page title for every event page. Of course, we can still override the title block in any child template. Template inheritance, you're awesome.

This 3-level inheritance is definitely not required, keep things simple if you can. But if you have many slightly different sections on your site, it might be perfect.

Route Prefix

Look back at the routing.yml file in our bundle. You're smart, so you probably already saw the prefix key and guessed that this prefixes all the imported route URLs with /event:

```
{# src/Yoda/EventBundle/Resources/config/routing.yml #}
{# ... #}

EventBundle_event:
    resource: "@EventBundle/Resources/config/routing/event.yml"
    prefix:   /event
```

This is a nice little feature. Now kill it!

```
{# src/Yoda/EventBundle/Resources/config/routing.yml #}
{# ... #}

EventBundle_event:
    resource: "@EventBundle/Resources/config/routing/event.yml"
    prefix:   /
```

With this gone, the events will show up on the homepage. Remove the /event from the URL in your browser to see it:

```
http://localhost:8000/app_dev.php
```

Chapter 11: Less Ugly with CSS and JavaScript

Things are still too ugly. I'll copy some CSS and image files I wrote up after the last chapter. We could put these in the web/ directory - it is publicly accessible afterall.

But there's a trick I want to show you, so let's copy them to a new Resources/public directory in EventBundle. Get these files by downloading the code for this screencast and looking in the resources directory. I already downloaded and copied that directory into my project for simplicity:

```
$ cp -r resources/public src/Yoda/EventBundle/Resources
```

Now we just need to add some link tags to base.html.twig. In fact, the layout already has a stylesheets block - let's put the link tags there:

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

{% block stylesheets %}
    <link rel="stylesheet" href="???" />
{% endblock %}
```

Why put them in a block? I'll show you exactly why in Episode 2, but basically this will let us include extra CSS files on only one page. The page-specific CSS file will show up *after* whatever we have in this block.

The assets:install Command

Wait a second - what should we put in the href? Only things in the web/ directory are web-accessible, and these *aren't* in there. What was I thinking?

Ok, so Symfony has a dead-simple trick here. Actually, it's console again, with its assets:install command. Get some help info about it first:

```
$ php app/console assets:install --help
```

As it says, the command copies the Resources/public directory from each bundle and moves it to a web/bundles directory. This little trick makes our bundle assets public!

And unless you're on windows, run this with the --symlink option: it creates a symbolic link instead of copying the directory:

```
$ php app/console assets:install --symlink
```

Now, our bundle's Resources/public directory shows up as web/bundles/event. There's even a few core bundles that use this trick.

assets:install with Composer

There's a secret. When we run php composer.phar install, the assets:install command is run automatically at the end. But it's not black-magic, there's just a scripts key in composer.json that tells it to do this and a few other things.

The uncool part about this is that it runs the command *without* the --symlink option. When the directories are copied instead of symlinked, testing CSS changes is a huge pain.

Edit the bottom of the composer.json script to activate the symlink option:

```
"extra": {
    " ... "
    "symfony-assets-install": "symlink",
},
```

The extra key is occasionally used in random ways like this. If you ever need to do anything else here, the README of some library will tell you.

The Twig asset Function

Ok, *now* lets finish up the link tags:

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

{% block stylesheets %}
    <link rel="stylesheet" href="{{ asset('bundles/event/css/event.css') }}" />
    <link rel="stylesheet" href="{{ asset('bundles/event/css/events.css') }}" />
    <link rel="stylesheet" href="{{ asset('bundles/event/css/main.css') }}" />
{% endblock %}
```

This is just the plain web path, except for the Twig asset function. This function doesn't do much, but it will make putting our assets on a CDN really easy later. So whenever you have a path to a CSS, JavaScript or image file, wrap it with this.

Preview to Assetic

This is cool. BUT, I want to give you a sneap peek of Assetic - a library that integrates with Symfony and lets you combine and process CSS and JS files:

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

{% block stylesheets %}
    {% stylesheets
        'bundles/event/css/event.css'
        'bundles/event/css/events.css'
        'bundles/event/css/main.css'
        filter='cssrewrite'
    %}
        <link rel="stylesheet" href="{{ asset_url }}" />
    {% endstylesheets %}
{% endblock %}
```

When we refresh, everything still looks the same. BUT, we've laid the foundation for being able to do things like use SASS and combining everything into 1 file for speed. We talk about Assetic more in Episode 4.

Chapter 12: Friendly Links and Dates in Twig

Let's look at a couple of things that are hiding in those new template files I created. Open up index.html.twig and notice the for tag, which loops over the entities variable we're passing to the template:

```
{# src/Yoda/EventBundle/Resources/views/Event/index.html.twig #}
{# ... #}

{% for entity in entities %}
    <article>...</article>
{% endfor %}
```

You're going to loop over a lot of things in your Twig days, so take a close look at the syntax.

Generating URLs

Further down, check out how we link to the "show" page for each event. Instead of hardcoding the URL, Symfony generates the URL based on the route:

```
{# src/Yoda/EventBundle/Resources/views/Event/index.html.twig #}
{# ... #}

<a href="{{ path('event_show', {'id': entity.id}) }}">
    {{ entity.name }}
</a>
```

Look at the event routes that were generated earlier and find one called event_show:

```
# src/Yoda/EventBundle/Resources/config/routing/event.yml
# ...

event_show:
    pattern:    /{id}/show
    defaults:   { _controller: "EventBundle:Event:show" }
```

To generate a URL in Twig, we use the Twig path function. Its first argument is the name of the route we're linking to, like event_show. The second is an array of wildcards in the route. We pass in the actual value we want for the id wildcard.

```
<a href="{{ path('event_show', {'id': entity.id}) }}">
    {{ entity.name }}
</a>
```

In the browser, you can see how each link generates almost the same URL, but with a different id portion.

Rendering Dates in a Template

One more hidden trick. The Event class's time field is represented internally by a PHP DateTime object. We saw this in our play.php file when we created an event. To actually render that as a string, we use Twig's date filter:

```
<dd>
    {{ entity.time|date('g:ia / l M j, Y') }}
</dd>
```

It transforms dates into another format and the string we pass to it is from PHP's good ol' fashioned date function formats.

Chapter 13: Adding Outside Bundles with Composer

We got rid of the ugly, but the site looks a little empty. We'll improve things by loading fixtures, which are dummy data we put into the database.

When we started the project, we downloaded the Symfony Standard edition: our pre-started project that came with Symfony and other tools like Doctrine. Unfortunately, it didn't come with any tools for handling fixtures.

But we're smart enough: let's just add a fixtures library ourselves. And by using Composer, doing this won't suck!

Head over to KnpBundles.com and search for "fixtures". Click on DoctrineFixturesBundle, yea, the one with the high quality score. Now click again to read its documentation.

Installing a Bundle via Composer

Composer is a PHP dependency management library. It downloads different libraries into our project and makes sure that their versions are all compatible with each other.

It works by reading the composer.json file inside your project. It downloads all of the libraries under the require key, *and* any libraries that *they* may depend on. To get the DoctrineFixturesBundle, copy the line from the documentation and paste it at the end of your require key:

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

Each library has two parts: its name and the version you want. The name comes from a site called Packagist.org. You can find almost any PHP library here and the versions available.

Finding the right Version

But using dev-master stinks. This tells Composer to grab the latest commit to the master branch, whatever craziness that may be.

Go back to the library's page on Packagist: anything without the dev at the end is a stable version. For me, the latest is 2.2.0. Let's use that, but add a ~ to the front of it:

```
{
    "require": {
        " ... ",
        "doctrine/doctrine-fixtures-bundle": "~2.2.0"
    }
}
```

With the tilde, this really means 2.2.*. Composer explains the different version formats really well on their site (Package Versions).

Installing with Composer

Ok, let's download this library! We'll need the composer.phar file from earlier - just move it into the project:

```
$ cp ../composer.phar .
```

And remember, this is just a normal file, so you can download as many of these as you want at GetComposer.org.

Now, run php composer.phar update and pass it the name of the library:

```
$ php composer.phar update doctrine/doctrine-fixtures-bundle
```

This may work for a little while as Composer things really hard about dependencies. Eventually, it'll download DoctrineFixturesBundle *and* its dependent doctrine-data-fixtures library into the vendor/ directory.

Composer update, install and composer.lock

While we wait, let's look at a small mystery. We know that Composer reads information from composer.json. So what's the purpose of the composer.lock file that's at the root of our project and how did it get there?

Composer actually has 2 different commands for downloading vendor stuff.

composer update

The first is update. It says "read the composer.json file and update everything to the latest versions specified in there". So if today we have Symfony 2.4.1 but 2.5.0 gets released, a Composer update would upgrade us to the new version. That's because our Symfony version constraint of ~2.4 allows for anything greater than 2.4, but less than 3.0.

Hold up. That could be a big issue. What happens if you deploy right as Symfony 2.5.0 comes out? Will your production server get that version, even though you were testing on 2.4.1? That would be *lame*.

Because Composer is *not* lame, each time the composer.phar update command is run, it writes a composer.lock file. This records the exact versions of all of your vendors at that moment.

composer install

And that's where the second command - install - comes in. It *ignores* the composer.json file and reads entirely from the composer.lock file, assuming one exists. So as long as you run install on your deploy, you'll get the exact versions you expected.

So unless you're adding a new library or intentionally upgrading something, always use composer.phar install.

And when you *do* need to add or update something, you can be more precise by calling composer.phar update and passing it the name of the library you're updating like we did. With this, Composer will only update *that* library, instead of everything.

Chapter 14: Fixtures: For some dumb data

We have the bundle! Plug it in! Open up the AppKernel class and add it there:

```
// app/AppKernel.php
// ...

$bundles = array(
    // ...
    new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle(),
);
```

To see if it's working, try getting help information on a new doctrine:fixtures:load console task that comes from the bundle:

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

We see the help information, so we're ready to write some fixtures.

Writing Fixtures

A fixture is just a PHP class that puts some stuff into the database.

Create a new file in the DataFixtures\ORM directory of your bundle. Let's call it LoadEvents.php, though the name doesn't matter.

Create an src/Yoda/EventBundle/DataFixtures/ORM/LoadEvents.php file.

To breathe life into this, copy and paste the example from the docs. Change the namespace above the class to match our project. Notice that the namespace always follows the directory structure of the file:

```
// src/Yoda/EventBundle/DataFixtures/ORM/LoadEvents.php
namespace Yoda\EventBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;

class LoadEvents implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        // .. todo
    }
}
```

Now we just use normal Doctrine code to create and save events. This is the play.php file all over again:

```
use Yoda\EventBundle\Entity\Event;
// ...

public function load(ObjectManager $manager)
{
    $event1 = new Event();
    $event1->setName('Darth\'s Birthday Party!');
    $event1->setLocation('Deathstar');
    $event1->setTime(new \DateTime('tomorrow noon'));
    $event1->setDetails('Ha! Darth HATES surprises!!!');
    $manager->persist($event1);

    $event2 = new Event();
    $event2->setName('Rebellion Fundraiser Bake Sale!');
    $event2->setLocation('Endor');
    $event2->setTime(new \DateTime('Thursday noon'));
    $event2->setDetails('Ewok pies! Support the rebellion!');
    $manager->persist($event2);

    // the queries aren't done until now
    $manager->flush();
}
```

Notice that we only need to call flush once. Doctrine prepares all of its work and then sends the queries as efficiently as possible all at once.

Loading the Fixtures

Ok, let's load some fixtures. Go back to the console and try the new doctrine:fixtures:load command:

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

When we look at the site, we've got fresh dummy data to play with. Re-run the command whenever you want to start over: it deletes everything and inserts the fixtures in a fresh state.

Tip

If you'd rather add to the existing data, just pass the --append option.

Chapter 15: Autoloading: Where did require/include go?

Autoloading: it's like plumbing. You forget it's there, but when it's gone, well, let's just say you have to go outside a bit more often.

Autoloading is the magic that lets us use classes without needing to require or include the file that holds them first. We used to have include statements everywhere, and well, it was terrible.

But an autoloader has a tricky job: given any class name, it needs to know the exact location of the file that holds that class. In many modern projects, including ours, Composer handles this, and there are two pieces to understanding how it figures out what file a class lives in.

Directory Structure and Namespaces

When we create an Event object, Composer's autoloader knows that this class lives inside src/Yoda/EventBundle/Entity/Event.php. How? It just takes the full class name, flips the slashes, and adds .php to the end of it:

```
Yoda\EventBundle\Entity\Event

src/Yoda/EventBundle/Entity/Event.php
```

As long as the namespace matches the directory and the class name matches the filename plus .php, autoloading just works. Let's mess this up - let's rename the Entity directory to Entity2:

```
use Yoda\EventBundle\Entity\Event;

// the src/Yoda/EventBundle/Entity/Event.php file is "included"
// .. so the file better exist and "house" the Event class!
new Event();
```

If we run play.php now, it fails big:

> Class Yoda\EventBundle\Entity\Event not found

The autoloader is looking for an Entity directory. Rename the directory back to Entity to fix things.

Library Directory Paths

Right now, it almost looks like the autoloader assumes that everything must live in the src/ directory. So how are vendor classes - like Symfony - loaded?

That's the second part. When we fetch a library with Composer, it configures its autoloader to look for the new classes in the directory it just downloaded.

Open up the vendor/composer/autoload_namespaces.php file. This is generated by Composer and it has a map of namespaces to the directories where those classes can be found:

```
// vendor/composer/autoload_namespaces.php
// ...

return array(
    'Symfony\\' => array($vendorDir . '/symfony/symfony/src'),
    // ...
    'Doctrine\\ORM' => $vendorDir . '/doctrine/orm/lib/',
    'Doctrine\\DBAL' => $vendorDir . '/doctrine/dbal/lib/',
    'Doctrine\\Common\\DataFixtures' => $vendorDir . '/doctrine/data-fixtures/lib/',
    // ...
);
```

So when we reference a Symfony class, it does the slash-flipping trick, and then looks for the file starting in vendor/symfony/symfony/src:

```
Symfony\Component\HttpFoundation\Response

vendor/symfony/symfony/src/Symfony/Component/HttpFoundation/Response.php
```

Now you know *all* the secrets about the autoloader. And when you see a class not found error, it's *your* fault. Sorry! The most common mistake is easily a missing use statement. If it's not that, check for a typo in your class and filename.

Chapter 16: Do Less Work in the Controller

Wow. You've already almost finished with first episode on Symfony2. That's not easy - Symfony2 has a learning curve, and I've been throwing a lot of tough concepts at you.

As a reward, let's see a few shortcuts!

The @Template Rendering Shortcut

Head to google and search for SensioFrameworkExtraBundle. Click the link on Symfony.com. This bundle is all about shortcuts, and it came standard with our project.

On the docs, scroll down to the @Template link and click it. Copy the use statement from the code block and paste it into EventController:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

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

Ok, let's remove the render method and just return the array of variables we were passing to the template. Finish up by adding an @Template annotation above indexAction with the template name:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

/**
 * @Template("EventBundle:Event:index.html.twig")
 */
public function indexAction()
{
    // ...

    return array(
        'entities' => $entities,
    );
}
```

When we check the page in our browser, it works perfectly. This is a bit of magic: it tells Symfony to render the template for us.

Now remove the template name and refresh:

```
/**
 * @Template()
 */
public function indexAction()
{
    // ...
}
```

This works too! Now we're talking.

If we don't pass a template name, it guesses it from the controller and action name:

```
Controller: EventBundle:Event:index

Template: EventBundle:Event:index.html.twig
```

Annotation use Statements

Let's talk about the use statement we pasted in. Whenever you use an annotation, you *must* have a use statement for it. If you don't, you'll see a nice exception:

> [SemanticalError] The annotation "@Template" in method .. was never imported. Did you maybe forget to add a "use" statement for this annotation?

We actually saw this already in our Event entity. It was generated for us, but it has a use statement for its ORM annotation.

Annotation Routing

What else can we do with annotations? How about routing?

On the docs, go back and click the @Route and @Method link. Copy the use statement into the controller and put an @Route annotation above indexAction:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class EventController extends Controller
{
    /**
     * @Template()
     * @Route("/")
     */
    public function indexAction()
    {
        // ...
    }
}
```

Importing Routes from a Controller

Hmm, routing right above the controller? Interesting!

But like all routing - we have to import it before it works. Open up the routing.yml file in the bundle, copy the event.yml import line and change the key so its unique. To import annotation routes, just point the resource at the Controller directory and add a type option:

```
# src/Yoda/EventBundle/Resources/config/routing.yml
# ...

EventBundle_event_annotation:
    resource: "@EventBundle/Controller"
    prefix:   /
    type:     annotation
```

When we refresh, it still works.

Duplicate Routes

BUT, things are not as they seem. Check out the web debug toolbar. It says that the event route is being matched. Now, run the router:debug console task. Uh oh, we have *two* routes with identical paths:

```
event                  ANY        ANY    ANY  /
...
yoda_event_event_index   ANY        ANY    ANY  /
```

The first route is from event.yml and the second is from our annotations where Symfony generates a name automatically by default. When two routes have the same path, the *first* route matches. So let's remove the first one in event.yml:

```
# src/Yoda/EventBundle/Resources/config/routing/event.yml
# event:
#     pattern:  /
#     defaults: { _controller: "EventBundle:Event:index" }
```

Now when we refresh, it works *and* our route is matched.

But when we try to create a new event, we get an error! In new.html.twig we're generating a link to the homepage by using its route name - event. Symfony generated a different name for the annotation route: yoda_event_event_index.

Easy fix. Just add a name="event" key to the routing annotation:

```
// src/Yoda/EventBundle/Controller/EventController.php
// ...

/**
 * @Template()
 * @Route("/", name="event")
 */
public function indexAction()
{
    // ...
}
```

And just like that, life is good. For homework, read through these docs and see what other cool things you can do.

Chapter 17: Twig Mind Tricks

I've got 2 more bonuses from Twig. In every template, you have access to a variable called app. This has a bunch of useful things on it, like the request, the security context, the User object, and the session:

```
{# src/Yoda/EventBundle/Resources/views/Event/index.html.twig #}
{# ... #}

{% block body %}
    {# some examples - remove these after you try them #}
    {{ app.session.get('some_session_key') }}
    {{ app.request.host }}

    {# ... #}
{% endblock %}
```

It's actually an object called GlobalVariables, which you can check out yourself. So when you need one of these things, remember app!

Tip

Remove this code after trying it out - it's just an example of how you can access the request and session data - it doesn't add anything real to our project.

The block Twig Function

Next, head to our base template. Right now, the title tag is boring: I can either replace it entirely in a child template or use the default:

```
<title>{% block title %}Welcome!{% endblock %}</title>
```

Let's make this better by adding a little suffix whenever the page title is overridden. This shows off the block function, which gives us the current value of a block:

```
<title>
    {% if block('title') %}
        {{ block('title') }} | Starwars Events
    {% else %}
        Events from a Galaxy, far far away
    {% endif %}
</title>
```

Refresh the events page to see it in action.

Twig Whitespace Control

But if you view the source, you'll see that we've got a lot of whitespace around the title tag. That's probably ok, but let's fix it anyways. By adding a dash to any Twig tag, all the whitespace on that side of the tag is removed. The end result is a title tag, with no whitespace at all:

```
<title>
    {%- if block('title') -%}
        {{ block('title') }} | Starwars Events
    {%- else -%}
        Events from a Galaxy, far far away
    {%- endif -%}
</title>
```

Wow! Congrats on finishing the first episode! You're well on your way with Symfony, so keep going with Episode 2 and start practicing on a project.

Seeya next time!