The Symfony Micro-Framework

With <3 from SymfonyCasts

Chapter 1: Bootstrapping Micro-Symfony

Inspiration strikes! You've just come up with a crazy awesome idea - the kind of idea that's sure to save the world and impress your friends all at once. Now all you need to do is start coding.

Of course, Symfony - the fully-featured full-stack framework will be perfect for this. Then again, what if we used Silex? The micro-framework approach might be a better way to get started so the world can quickly find out what its been missing!

But Silex isn't Symfony: it lacks a lot of the tools. And if your project grows, it can't easily evolve into a Symfony app: they're just too different.

There's another option: use Symfony as a micro-framework, meaning with as few files, directories and complications as possible, without sacrificing features. And since no official Symfony micro-framework exists right now, let's create one.

Composer Bootstrap

Start with a completely empty directory: so empty that we need to run composer init to bootstrap a composer.json file:

```
$ composer init
```

Fill in the name - none of this matters too much in a private project. Ok, dependencies. We only need one: symfony/symfony. But I also want one more library: sensio/framework-extra-bundle - this is for annotation routing. If you don't want that, you only need symfony/symfony.

Our blank project now has 1 file - composer.json - with just 2 dependencies:

```
14 lines | composer.json
{
    "name": "knpuniversity/micro-symfony",
    "require": {
        "symfony/symfony": "^2.7",
        "sensio/framework-extra-bundle": "^3.0"
    },
... lines 7 - 12
}
```

Go back to run composer install:

```
$ composer install
```

Now our empty project has a vendor/ directory.

Bootstrapping Symfony

Now that we have Symfony how do we bootstrap the framework? We need two things: a kernel class and a front controller script that boots and executes it.

Start with the kernel: create a new AppKernel.php file at the root of your project. You can call this file anything, but I want to stay consistent with normal Symfony when possible. Make this extend Symfony's Kernel class. Since we're not in a namespaced class, PhpStorm auto-completes the namespaces inline. Ew! Add a use statement instead: let's keep things somewhat organized people:

```
24 lines | AppKernel.php
... lines 1 - 2
use Symfony\Component\HttpKernel\Kernel;
... lines 4 - 5
class AppKernel extends Kernel
{
... lines 8 - 22
}
```

Kernel is an abstract class. Use cmd+n - or the menu option, Code->Generate - then "Implement Methods". Select the two methods at the bottom:

```
24 lines | AppKernel.php
... lines 1 - 2
use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Config\Loader\LoaderInterface;

class AppKernel extends Kernel
{
    public function registerBundles()
    {
... lines 10 - 16
    }

    public function registerContainerConfiguration(LoaderInterface $loader)
    {
... line 21
    }
}
```

registerBundles()

Get rid of the comments. So first, let's instantiate all the bundles we need. In a normal, new, Symfony project, there are 8 or more bundles here by default. We just need 3: a new FrameworkBundle, TwigBundle - this is used to render error and exception pages, so you may want this, even if you're building an API. And since I'll use annotations for my routing, use SensioFrameworkExtraBundle. If you like YAML routing, you don't even need this last one. Return the $bundles:

```
24 lines | AppKernel.php
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new \Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            new \Symfony\Bundle\TwigBundle\TwigBundle(),
            new \Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        );

        return $bundles;
    }
... lines 18 - 22
}
```

registerContainerConfiguration()

For the next method - move the inline namespace to be a use statement on top. When the kernel boots, it needs to configure each bundle - it needs things like the database password, where to log, etc. That's the purpose of registerContainerConfiguration().

Use $loader->load() and point this to a new config/config.yml file:

```
24 lines | AppKernel.php
... lines 1 - 5
class AppKernel extends Kernel
{
... lines 8 - 18
    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(__DIR__.'/config/config.yml');
    }
}
```

And now create the config directory and that file so that it isn't pointing into outerspace. Leave it blank for now.

Front Controller: index.php

That's a functional kernel! We need a front controller that can instantiate and execute it. Create a web/ directory with an index.php inside. Symfony has two front controllers: app.php and app_dev.php: we'll have just one because I want less less less. We're going for the micro of micro!

In my browser, we're looking at the symfony/symfony-standard repository, because I want to steal some things. Open the web/ directory and find app_dev.php. Copy its contents into index.php. Now we're going to slim this down a bit:

31 lines | web/index.php

```
<?php

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Debug\Debug;

// If you don't want to setup permissions the proper way, just uncomment the following PHP line
// read http://symfony.com/doc/current/book/installation.html#configuration-and-setup for more information
//umask(0000);

// This check prevents access to debug front controllers that are deployed by accident to production servers.
// Feel free to remove this, extend it, or make something more sophisticated.
if (isset($_SERVER['HTTP_CLIENT_IP'])
    || isset($_SERVER['HTTP_X_FORWARDED_FOR'])
    || !(in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', 'fe80::1', '::1')) || php_sapi_name() === 'cli-server')
) {
    header('HTTP/1.0 403 Forbidden');
    exit('You are not allowed to access this file. Check '.basename(__FILE__).' for more information.');
}

$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();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

Delete the IP-checking stuff and uncomment out the umask - that makes dealing with permissions easier. For the loader, require Composer's standard autoloader in vendor/autoload.php. And AppKernel is *not* in an app/ directory - so update the path:

24 lines | web/index.php

```
... lines 1 - 4
umask(0000);
... lines 6 - 10
$loader = require_once __DIR__.'/../vendor/autoload.php';
require_once __DIR__.'/../AppKernel.php';
... lines 13 - 24
```

When you create the kernel, you pass it an environment and whether or not we're in debug mode: that controls whether errors are shown. We'll tackle these properly soon, but right now, hardcode two new variables $env = 'dev' and $debug = true. Pass these into AppKernel:

24 lines | web/index.php

```
... lines 1 - 7
$env = 'dev';
$debug = true;
... lines 10 - 17
$kernel = new AppKernel($env, $debug);
... lines 19 - 24
```

Oh, and put an if statement around the Debug::enable() line so it only runs in debug mode. This wraps PHP's error handler and gives us better errors:

24 lines | web/index.php

```
... lines 1 - 13
if ($debug) {
    Debug::enable();
}
... lines 17 - 24
```

And in the spirit of making everything as small as possible, remove the loadClassCache() line: this adds a small performance boost. If you're feeling less micro, you can keep it.

Testing it out

That's it for the front controller! Time to try things. Open up a new terminal tab, move into the web directory, then start the built-in web server on port 8000:

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

Let's try it - we're hoping to see a working Symfony 404 page, because we don't have any routes yet. Okay, not working yet - but this isn't so bad: it's a Symfony error, we're missing some kernel.secret parameter.

Adding the Base Config

The reason is that we do need a *little* bit of configuration in config.yml. Add a framework key with a secret under it that's set to anything:

7 lines | config/config.yml

```
framework:
    secret: ABC123
... lines 3 - 7
```

When we refresh now, it's a different error: something about a Twig template that's not defined. These weird errors are due to some missing configuration. Here's the minimum you need. Add a router key with a resource sub-key that points to a routing file. Use %kernel.root_dir%/config/routing.yml. %kernel.root_dir% points to wherever your kernel lives, which is the app/ directory in normal Symfony, but the root folder for us minimalists.

You also need a templating key with an engines sub-key set to twig:

7 lines | config/config.yml

```
framework:
    secret: ABC123
    router:
        resource: '%kernel.root_dir%/config/routing.yml'
    templating:
        engines: [twig]
```

That's all you need to get things working. Oh, and don't forget to create a config/routing.yml file - it can be blank for now:

1 lines | config/routing.yml

Refresh! It lives! "No route found for GET /" - that's our working 404 page. We'll fix that missing CSS in a bit, but this *is* a functional Symfony project: it just doesn't have any routes yet.

The Symfony Plugin

One of the coolest things about using Symfony is the PhpStorm plugin for it, which we cover in the Lean and Mean dev with PHP Storm tutorial.

If you have it installed, search "symfony" in your settings to find it. Check "Enable" and remove app/ from all the configurable paths. Our project is smaller than normal: instead of app/cache and app/logs, you just have cache and logs at the root. Init a new git repository and add these to your .gitignore file along with vendor/:

4 lines | .gitignore

```
/vendor/
/cache/
/logs/
```

Count the files: other than composer stuff, we have 1, 2, 3, 4 files, and a functioning Symfony framework project. #micro

Chapter 2: AppBundle, Routing and Annotations

Now, we need to build a route and create a page. But I want to do this with as few files as possible. Remember, we're going for micro here!

In Symfony, your PHP code lives in a bundle. But actually, you don't *need* a bundle at all: you could code entirely without one. And this could give us one less file.

In practice, *not* having a bundle complicates a few things. So, we *will* create one here, but if you're curious about the approach, ask me in the comments, or read our AppBundle blog post that talks about this.

Creating AppBundle

Create a new directory called just AppBundle - not src/AppBundle - I want to eliminate some directories! Put a new PHP class in there *also* called AppBundle in an AppBundle namespace. Make this class extend Bundle:

10 lines | AppBundle/AppBundle.php

```
... lines 1 - 2
namespace AppBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class AppBundle extends Bundle
{
}
```

Activate this in AppKernel with new AppBundle/AppBundle():

25 lines | AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 13
            new \AppBundle\AppBundle()
        );
... lines 16 - 17
    }
... lines 19 - 23
}
```

Routing

Cool - that's the last time you'll need to do that. Now let's create a page. Open routing.yml. To minimize the number of files I need to touch I'll use annotation routing. If you prefer YAML, you can start adding your routes here, point them to controller classes and be on your way.

For us, add an import: resource: @AppBundle/Controller with type: annotation:

4 lines | config/routing.yml

```yaml
app_annotations:
  resource: "@AppBundle/Controller"
  type: annotation
```

In AppBundle, create the Controller directory and a new class in there called MighyMouseController in honor of another small, but powerful thing.

18 lines | AppBundle/Controller/MightyMouseController.php

```php
... lines 1 - 2
namespace AppBundle\Controller;
... lines 4 - 7
class MightyMouseController
{
... lines 10 - 16
}
```

Add public function rescueAction() and return a new Response(): "Here I come to save the day!":

18 lines | AppBundle/Controller/MightyMouseController.php

```php
... lines 1 - 4
use Symfony\Component\HttpFoundation\Response;
... lines 6 - 7
class MightyMouseController
{
... lines 10 - 12
    public function rescueAction()
    {
        return new Response('Here I come to save the day!');
    }
}
```

For routing, it's like normal: add the use statement for Route. Then above the method, finish things with @Route("/"):

18 lines | AppBundle/Controller/MightyMouseController.php

```php
... lines 1 - 5
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
... line 7
class MightyMouseController
{
    /**
     * @Route("/")
     */
    public function rescueAction()
    {
... line 15
    }
}
```

Autoloading!

Let's see if things work. Error! Where are you when we need you Mighty Mouse?

```
Attempted to load class "AppBundle" from namespace AppBundle.
```

The autoloader doesn't know that we have classes living inside the AppBundle directory. In a traditional Symfony project, there's a pre-made autoload section in composer.json that takes care of this for us. We on the other hand need to do this manually.

Add an autoload section, with a psr-4 section below it. Inside, we need just one entry AppBundle\\ mapped to AppBundle/:

```
19 lines | composer.json
{
... lines 2 - 12
    "autoload": {
        "psr-4": {
            "AppBundle\\": "AppBundle/"
        }
    }
}
```

This says that if a class's namespace starts with AppBundle, look for it inside the AppBundle directory. After that, it follows the same namespace rules as always. You can even rename AppBundle to something else like src - this is the only line you'd need to change.

To get the change to take effect, run composer dump-autoload:

```
$ composer dump-autoload
```

You don't normally need to run this - it happens after a composer install or update.

Autoloading Annotations

Try it again. Excellent - the next error!

```
The annotation "@Sensio\Bundle\FrameworkExtraBundle\Configuration\Route"
... does not exist.
```

It doesn't see the Route annotation class - it's almost as if we forgot the use statement. If you use annotations in a project, you need one extra autoload line.

In the config/ directory, create an autoload.php file. Go back to the Standard Edition code so we can cheat off of it. Find app/autoload.php and copy its contents. Paste that here:

```
8 lines | config/autoload.php
... lines 1 - 8
```

I'll delete some comments to make things really small. This includes the normal autoloader, then adds an extra thing for annotations. Now, in index.php - or anywhere else you need to autoload - require *this* file instead:

```
24 lines | web/index.php
... lines 1 - 10
$loader = require_once __DIR__.'/../config/autoload.php';
... lines 12 - 24
```

Refresh! Very nice! Let's count the files: 1, 2, 3, 4, 5, 6, 7. We're rocking the Symfony framework with basically less than 10 files.

Chapter 3: Container, Twig: All Smooth

When you want to render a template, you'll usually extend Controller and it'll give you a bunch of shortcuts, including render(). That's a great idea. But I'll extend ContainerAware instead:

24 lines | AppBundle/Controller/MightyMouseController.php

```php
... lines 1 - 4
use Symfony\Component\DependencyInjection\ContainerAware;
... lines 6 - 8
class MightyMouseController  extends ContainerAware
{
... lines 11 - 22
}
```

That gives access to the service container, but not the shortcut methods. So I can't say $this->render(), but I *can* say $html = $this->container->get('twig') and then call render() on it. Render mighty_mouse/rescue.html.twig and pass it a quote variable. But that's just HTML - so wrap it in a Response and return:

24 lines | AppBundle/Controller/MightyMouseController.php

```php
... lines 1 - 8
class MightyMouseController  extends ContainerAware
{
... lines 11 - 13
    public function rescueAction()
    {
        $html = $this->container->get('twig')->render(
            'mighty_mouse/rescue.html.twig',
            array('quote' => 'Here I come to save the day!')
        );

        return new Response($html);
    }
}
```

Now, where do templates live? Usually, it's app/Resources/views. But we've moved everything up and *out* of app/. So our path is just Resources/views: add those directories and a mighty_mouse folder inside. Create rescue.html.twig and extend the base template. We don't have a base template yet, but we will soon. In {% block body %} print out the quote:

6 lines | Resources/views/mighty_mouse/rescue.html.twig

```twig
{% extends 'base.html.twig' %}

{% block body %}
    <h1>{{ quote }}</h1>
{% endblock %}
```

Now add base.html.twig in Resources/views. Let's steal again from the Standard Edition. Find app/Resources/views/base.html.twig, copy it, and paste it in ours:

13 lines | 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>
```

Ok, give it a try. Unable to find template "mighty_mouse/rescue.html.twig". Hmm I bet I have a typo. Yep, my directory was called might_mouse. I went a bit too micro on the name there. Fix that and try again. It's alive!

Tip

After creating the views/ directory for the first time, you'll need to clear your Symfony cache, or the template won't be found. That's probably a small Symfony bug, that hopefully we can fix!

Now, let's add the web debug toolbar.

Chapter 4: Web Debug Toolbar and Profiler

Right now, there's no floating web debug toolbar on our pages. That's one of the best features of Symfony, and so I want it in my micro-edition too.

Instead of getting a lot of features automatically, we need to enable things one-by-one when and if we need them. The same is true for the web debug toolbar: it comes from the WebProfilerBundle. This lives in our project, but it's not enabled yet. Add it in AppKernel down below the others:

27 lines | AppKernel.php

```php
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
... lines 10 - 16
        $bundles[] = new \Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
... lines 18 - 19
    }
... lines 21 - 25
}
```

Next, head into config.yml - we need a little bit of configuration. Add a web_profiler key with a toolbar option. Set this to %kernel.debug%:

12 lines | config/config.yml

```yaml
... lines 1 - 9
web_profiler:
    toolbar: %kernel.debug%
```

That's a special parameter that's equal to the debug argument we pass to AppKernel in index.php:

24 lines | web/index.php

```php
... lines 1 - 8
$debug = true;
... lines 10 - 17
$kernel = new AppKernel($env, $debug);
... lines 19 - 24
```

Mostly, this flag is used to hide or show errors. But for now, we'll also use it to decide if the web debug toolbar should show up or not.

Also, under framework, add a profiler key with an enabled option that we'll also set to %kernel.debug%:

12 lines | config/config.yml

```yaml
framework:
... lines 2 - 6
    profiler:
        enabled: %kernel.debug%
... lines 9 - 12
```

We have debug set to true, so try it! No toolbar yet - just this nice error:

```
Unable to generate a URL for the named route "_wdt" as such
route does not exist.
```

This is the error you usually get if you're generating a URL to a route, but you're using a bad route name. The web debug toolbar and profiler are fueled by real Symfony routes, and those routes need to be included for this all to work. On the Standard Edition, I already have the app/config/routing_dev.yml file open, because we need to copy the top two entries. Find config/routing.yml and paste those there:

```yaml
// 12 lines | config/routing.yml
// ... lines 1 - 4
_wdt:
    resource: "@WebProfilerBundle/Resources/config/routing/wdt.xml"
    prefix:   /_wdt

_profiler:
    resource: "@WebProfilerBundle/Resources/config/routing/profiler.xml"
    prefix:   /_profiler
```

Try one more time. There's the toolbar, with all of the goods we love.

Chapter 5: dotenv: Environmental Variables

The $debug and $environment variables are hardcoded in index.php. So if we deploy this, we'll either have a web debug toolbar on production, or we'll need to manually modify this file to set $debug to false. Both situations stink.

PHP dotenv Fanciness

To help, we'll install a new library: composer require vlucas/phpdotenv:

```
$ composer require vlucas/phpdotenv
```

While we're waiting, Google for that library and find its documentation.

This is a popular library these days, and I really like it too. It allows you to have a .env file at the root of your project. It'll look something like this:

```
S3_BUCKET="dotenv"
SECRET_KEY="souper_seekret_key"
```

This library reads these values and turns S3_BUCKET and SECRET_KEY into environment variables. Then, in our app, whenever we need some configuration - like whether we're in debug mode, the database password or the S3 bucket - we'll read from the environment variables. When you eventually deploy, you can set those variables in two different ways. First, you can have a .env file. Or second, you can set the variables via something like your web server configuration. Some platforms - like Heroku also have a way to set environment variables. The important point is that your app isn't bound to a configuration file: it's just reading environment variables, which is a pretty standard way of setting config.

Our .env File

The first things we want to set are the $env and $debug flags. Create a .env file and say SYMFONY_ENV=dev and SYMFONY_DEBUG=1:

4 lines | .env

```
# basic setup
SYMFONY_ENV=dev
SYMFONY_DEBUG=1
```

Remove the old variables in index.php. Replace it with $dotenv = new DotEnv\DotEnv(). The argument is the directory where the .env file lives - it's actually up one directory from here. Then, call $dotenv->load():

26 lines | web/index.php

```
... lines 1 - 9
10 // load the environmental variables
11 $dotenv = new Dotenv\Dotenv(__DIR__.'/../');
12 $dotenv->load();
... lines 13 - 26
```

At this point, those two flags have been set as environment variables. That means we can say $env = $_SERVER['SYMFONY_ENV']; and $debug = $_SERVER['SYMFONY_DEBUG'];:

26 lines | web/index.php

```
... lines 1 - 9
// load the environmental variables
$dotenv = new Dotenv\Dotenv(__DIR__.'/../');
$dotenv->load();
$env = $_SERVER['SYMFONY_ENV'];
$debug = $_SERVER['SYMFONY_DEBUG'];
... lines 15 - 26
```

Go back and refresh the new setup - we should still see the toolbar. Yep, there it is. But now go back to .env and set SYMFONY_DEBUG to 0. Because of config.yml, this should turn the toolbar off. Change the environment to prod too - that's not being used anywhere yet, but it may avoid a temporary cache error:

4 lines | .env

```
# basic setup
SYMFONY_ENV=prod
SYMFONY_DEBUG=0
```

Try it out: no web debug toolbar.

Add .env to the .gitignore file - this shouldn't be committed:

5 lines | .gitignore

```
... lines 1 - 3
/.env
```

But copy .env to .env.example - we will commit this so that new developers have something they can use as a guide.

Chapter 6: Environments

We're kind of abusing the debug flag: it's mostly used to tell Symfony if it should hide/show errors. If you want to control how your app behaves while developing versus on production, that's normally done with environments: the prod environment will turn on caching and turn off debugging tools. The dev environment will do the opposite.

When debug is false, the web debug toolbar is basically off, but the WebProfilerBundle is still being loaded. That's a bit wasteful: why not only instantiate this in the dev environment?

Add if $this->getEnvironment() == 'dev': let's only load the bundle in that scenario:

48 lines | AppKernel.php

```
... lines 1 - 6
class AppKernel extends Kernel
{
    public function registerBundles()
    {
... lines 11 - 17
        if ($this->getEnvironment() == 'dev') {
            $bundles[] = new \Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        }
... lines 21 - 22
    }
... lines 24 - 46
}
```

Change .env back to ENV equals dev and DEBUG equals 1. Refresh now: no issues.

Now change the environment to prod and refresh. Ah, error:

```
There is no extension able to load the configuration for web_profiler.
```

This makes sense: now that the WebProfilerBundle is *not* being loaded in the prod environment, Symfony doesn't know how to handle the web_profiler section in config.yml. The full stack Symfony framework solves this by having environment-specific config_dev.yml and config_prod.yml files. But I've gotta keep things small. So, I'll show you a trick.

Conditional Configuration with a Trick

Find registerContainerConfiguration() and add a new $isDevEnv variable that uses the same getEnvironment() method from before. Next, use $loader->load() to load more configuration. But instead of passing it a file, pass it a Closure that accepts a ContainerBuilder argument. Add the use statement on top. Also add a use on the Closure so we have acccess to $isDevEnv:

```
48 lines | AppKernel.php
... lines 1 - 4
use Symfony\Component\DependencyInjection\ContainerBuilder;
... line 6
class AppKernel extends Kernel
{
... lines 9 - 24
    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(__DIR__.'/config/config.yml');

        $isDevEnvironment = $this->getEnvironment() == 'dev';

        // do some dynamic customizations
        $loader->load(function (ContainerBuilder $container) use ($isDevEnvironment) {
... lines 33 - 44
        });
    }
}
```

Now when Symfony loads, it'll load everything from config.yml, but it'll also call this function, and we can control whatever extra configuration we want. In config.yml, we can't have the web_profiler stuff because this file is *always* loaded. Remove that:

[[code('7dc1f3ad36')]]

But inside the Closure, we can say: if we're in the dev environment, then call $container->loadFromExtension() to pass in the configuration we want. Use web_profiler as the first agument and an array as the second with toolbar => true:

```
48 lines | AppKernel.php
... lines 1 - 6
class AppKernel extends Kernel
{
... lines 9 - 24
    public function registerContainerConfiguration(LoaderInterface $loader)
    {
... lines 27 - 28
        $isDevEnvironment = $this->getEnvironment() == 'dev';
... lines 30 - 31
        $loader->load(function (ContainerBuilder $container) use ($isDevEnvironment) {
            if ($isDevEnvironment) {
                $container->loadFromExtension('web_profiler', array(
                    'toolbar' => true,
                ));
            }
... lines 38 - 44
        });
    }
}
```

That should take care of the first error, so refresh. Closer - that error is gone, but now there's an error loading the wdt.xml routing file. Again, this makes sense: we're still loading some routes from @WebProfilerBundle, which simply doesn't exist in the prod environment:

12 lines | config/routing.yml

```yaml
... lines 1 - 4
_wdt:
    resource: "@WebProfilerBundle/Resources/config/routing/wdt.xml"
    prefix:   /_wdt

_profiler:
    resource: "@WebProfilerBundle/Resources/config/routing/profiler.xml"
    prefix:   /_profiler
```

To fix this, create a routing_dev.yml file, move the two WebProfilerBundle imports there, and at the bottom, import the main routing.yml file:

13 lines | config/routing_dev.yml

```yaml
# this is "dev-only" debugging routes:
# you probably don't want to put anything else in this file
_wdt:
    resource: "@WebProfilerBundle/Resources/config/routing/wdt.xml"
    prefix:   /_wdt

_profiler:
    resource: "@WebProfilerBundle/Resources/config/routing/profiler.xml"
    prefix:   /_profiler

_main:
    resource: routing.yml
```

Now, do you remember where we told Symfony to load routing.yml? It's in config.yml:

12 lines | config/config.yml

```yaml
framework:
... line 2
    router:
        resource: '%kernel.root_dir%/config/routing.yml'
... lines 5 - 12
```

In *all* cases, we're loading routing.yml. But now we need to do something conditional: in dev, we want to load routing_dev.yml. Otherwise, we want to load routing.yml. And again, we can do this in AppKernel: this is the spot you'll go to if you need to configure something environment-specific.

If we're in the dev environment, call $container->loadFromextension(). This time, we need to tweak the framework configuration to override the router.resource value. In the array, add a router key, set it to an array, then add resource and set it to routing_dev.yml:

```
49 lines | AppKernel.php
... lines 1 - 6
class AppKernel extends Kernel
{
... lines 9 - 24
    public function registerContainerConfiguration(LoaderInterface $loader)
    {
... lines 27 - 28
        $isDevEnvironment = $this->getEnvironment() == 'dev';
... lines 30 - 31
        $loader->load(function (ContainerBuilder $container) use ($isDevEnvironment) {
... lines 33 - 38
            if ($isDevEnvironment) {
                $container->loadFromExtension('framework', array(
                    'router' => array(
                        'resource' => '%kernel.root_dir%/config/routing_dev.yml',
                    )
                ));
            }
        });
    }
}
```

In every environment, we load config.yml. But in the dev environment, we *override* the router resource config.

Remember, we're in the prod environment. Refresh. No errors, no web debug toolbar. Change the environment back to dev and refresh. Whoops: it doesn't like my webprofiler configuration because that's a typo! Make sure you have web_profiler:

```
49 lines | AppKernel.php
... lines 1 - 24
    public function registerContainerConfiguration(LoaderInterface $loader)
    {
... lines 27 - 33
                $container->loadFromExtension('web_profiler', array(
                    'toolbar' => true,
                ));
... lines 37 - 46
    }
... lines 48 - 49
```

Now it's happy, I'm happy *and* we have the toolbar.

The power of the environment is that we can have two different sets of configuration right on one machine, and toggling between them is effortless. But to minimize files, we'll keep all the environmental differences inside this Closure.

Parameters in dotenv

There's one more trick to configuration. In config.yml we have a secret key. This should be big, long, random and - of course - secret. It should *not* be committed to the repository. Normally, we'd set this to a parameter - like %secret%:

```
9 lines | config/config.yml
framework:
    secret: %secret%
... lines 3 - 9
```

Then, we'd add this to our parameters.yml file and be done. But no more! We can do this in .env.

If I add SECRET = CHANGEME, that'll create an environment variable called SECRET, but it will not automatically create a

Symfony parameter called secret. If I refresh, we see the error:

```
You have requested a non-existent parameter "secret".
```

Normally, parameters are created by adding a parameters key in any configuration file and listing them below. But we *don't* want to put this stuff here.

The secret is that if you prefix any environment variable with SYMFONY__, Symfony will take what's after that and automatically turn that into a parameter:

```
7 lines | .env
... lines 1 - 4
# parameters (SYMFONY__ is mapped to parameters)
SYMFONY__SECRET=CHANGEME
```

Refresh. Now it works perfectly. Goodbye parameters.yml, hello .env with SYMFONY__ used as the prefix.

And don't forget to put this line in the .env.example file so new developers have a chance to get things setup:

```
7 lines | .env.example
... lines 1 - 4
# parameters (SYMFONY__ is mapped to parameters)
SYMFONY__SECRET=CHANGEME
```

Chapter 7: Where's my app/console

We have environments and everything is going great, but we don't have an app/console script. So we can't use any of the handy commands like debug:router or debug:container. *And*, if you surf to a 404 page, it's still missing its CSS. That's because we need to run app/console assets:install so that it can symlink or copy a few core CSS files that the exception page uses. That's a bit difficult right now, since there is no console script in our project.

Let's go steal it from the standard edition. On GitHub, head into the app directory, open console and copy its contents. Now, put our new console file right at the root of the project and make sure PhpStorm formats it as a PHP file. Paste the contents:

28 lines | console

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


// if you don't want to setup permissions the proper way, just uncomment the following PHP line
// read http://symfony.com/doc/current/book/installation.html#configuration-and-setup for more information
//umask(0000);

set_time_limit(0);

require_once __DIR__.'/bootstrap.php.cache';
require_once __DIR__.'/AppKernel.php';

use Symfony\Bundle\FrameworkBundle\Console\Application;
use Symfony\Component\Console\Input\ArgvInput;
use Symfony\Component\Debug\Debug;

$input = new ArgvInput();
$env = $input->getParameterOption(array('--env', '-e'), getenv('SYMFONY_ENV') ?: 'dev');
$debug = getenv('SYMFONY_DEBUG') !== '0' && !$input->hasParameterOption(array('--no-debug', '')) && $env !== 'prod';

if ($debug) {
    Debug::enable();
}

$kernel = new AppKernel($env, $debug);
$application = new Application($kernel);
$application->run($input);
```

Let's go to work: uncomment the umask. For the first require_once, we know that this should be config/autoload.php. And AppKernel lives right in this directory, so that's fine:

30 lines | console

```
... lines 1 - 3
umask(0000);
... lines 5 - 7
require_once __DIR__.'/config/autoload.php';
require_once __DIR__.'/AppKernel.php';
... lines 10 - 30
```

The biggest thing we need to change is the $env and $debug variables. Let's grab that code from index.php. PhpStorm is being weird and hiding my web/ directory... now it's back. Copy the 4 dotenv lines from index.php. Now in console, delete the $env and $debug lines and paste our stuff:

```
30 lines | console
... lines 1 - 14
// load the environmental variables
$dotenv = new Dotenv\Dotenv(__DIR__);
$dotenv->load();
$env = $_SERVER['SYMFONY_ENV'];
$debug = $_SERVER['SYMFONY_DEBUG'];
... lines 20 - 30
```

Instead of passing a --env=prod flag to change the environment, this reads the .env file like normal. Let's try it!

If you're on UNIX, make this executable with a chmod +x. Then run ./console or in Windows php console:

```
$ chmod +x console
$ ./console
```

That blows up with "Environment file .env not found". I forgot to update my DotEnv path since this file lives in the root. Make sure you pass it just __DIR__:

```
30 lines | console
... lines 1 - 15
$dotenv = new Dotenv\Dotenv(__DIR__);
... lines 17 - 30
```

Try it now:

```
$ ./console
```

Our mini-app has a fully-fledged console. Instead of ./app/console, the command is just ./console, but everything else is the same. Use debug:router to get all the routes:

```
$ ./console debug:router
```

And debug:container to see the services:

```
$ ./console debug:container
```

And the one we want is ./console assets:install with a --symlink if your system supports that.

```
$ ./console assets:install --symlink
```

Refresh the 404 page: there's the beautiful exception page we expect.

Chapter 8: Logging and Adding other Tools

Our micro-Symfony is done: go forth and use this as the starting point for new projects. It's fully-functional, it *looks* a lot like normal Symfony, but it's just a lot smaller. Mission accomplished.

As you build your project, you may start missing some features that you normally get out of the box with Symfony, like logging - we don't have *any* logging capabilities right now.

Symfony logging usually comes from MonologBundle: we can look for it in our project, but nope - we don't have MonologBundle yet. That's no problem: we just need to treat this normally-core bundle like any other third-party bundle.

Enabling and Configuring MonologBundle

So, step 1 is to install it. At your terminal, run composer require symfony/monolog-bundle:

```
$ composer require symfony/monolog-bundle
```

While we're waiting, Google for "symfony monologbundle". The first link is to its GitHub page, which basically just points you to read the official documentation on Symfony.

Normally, Symfony users already have this bundle installed in their project. So we need to do 3 extra steps. First, we needed to grab it with composer. Done! Second, we need to enable it in AppKernel: new MonologBundle:

50 lines | AppKernel.php

```
... lines 1 - 6
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 12 - 14
            new \Symfony\Bundle\MonologBundle\MonologBundle(),
... line 16
        );
... lines 18 - 23
    }
... lines 25 - 48
}
```

And third - if required - we need to configure the bundle. For that, go back one more time to the Standard Edition and find the app/config/config_prod.yml file. Copy most of its monolog configuration, it's good stuff! Open up our config.yml and paste it in. Change action_level to debug to log everything.

20 lines | config/config.yml

```yaml
    ... lines 1 - 9
monolog:
    handlers:
        main:
            type:         fingers_crossed
            action_level: debug
            handler:      nested
        nested:
            type:  stream
            path:  "%kernel.logs_dir%/%kernel.environment%.log"
            level: debug
```

Refresh! No errors. *And* we suddenly have a logs directory with a dev.log file in it. Run tail -f on that to watch it:

```
$ tail -f logs/dev.log
```

Clear the screen and refresh. Great - it's logging *a lot* of things.

Controlling action_level

We'll probably *not* want to log *everything* on production like this. So, we need a way to control the action_level. Let's use a parameter: set it to %log_action_level%:

20 lines | config/config.yml

```yaml
    ... lines 1 - 9
monolog:
    handlers:
        main:
    ... line 13
            action_level: %log_action_level%
    ... lines 15 - 20
```

And of course if we try it now, we see "non-existent parameter log_action_level". How can we add it? You already know: go to .env and add SYMFONY__LOG_ACTION_LEVEL: Symfony will lowercase that before making it a parameter. Set it to debug:

8 lines | .env

```
    ... lines 1 - 6
SYMFONY__LOG_ACTION_LEVEL=debug
```

And immediately copy this line into .env.example:

8 lines | .env.example

```
    ... lines 1 - 6
SYMFONY__LOG_ACTION_LEVEL=debug
```

Clear the logs again and refresh. They're still *very* verbose. Change the level to error in .env.

Clear the logs and refresh. Hey, *nothing* in the logs: there weren't any errors. Try a 404 page. Perfect, *all* the logs from the request show up as expected. That's the job of the fingers_crossed handler: don't show me any logs, unless there's at least one entry that's an error level or higher. Then I want everything.

Enabling dump()

Ok, there's one more thing I love that I'm missing. In the controller, add a dump($this->container->get('twig')):

```
26 lines | AppBundle/Controller/MightyMouseController.php
... lines 1 - 8
class MightyMouseController  extends ContainerAware
{
... lines 11 - 13
    public function rescueAction()
    {
... lines 16 - 20
        dump($this->container);
... lines 22 - 23
    }
}
```

That's the new awesome dump() function from Symfony 2.6. Right now, it gives us an UndefinedFunctionException:

```
Attempted to call function `dump()`.
```

Ok, this works in Symfony normally, where does it come from? It comes from a DebugBundle. Head to AppKernel and enable it in the dev environment: new DebugBundle():

```
51 lines | AppKernel.php
... lines 1 - 6
class AppKernel extends Kernel
{
    public function registerBundles()
    {
... lines 11 - 18
        if ($this->getEnvironment() == 'dev') {
            $bundles[] = new \Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
            $bundles[] = new \Symfony\Bundle\DebugBundle\DebugBundle();
        }
... lines 23 - 24
    }
... lines 26 - 49
}
```

Try it again. The page loads, and in the web debug toolbar, we see the dumped object.

This *is* Symfony, but it's smaller: you need to enable things when you want them. The minimum setup includes 4 files in config, 1 in web, and 3 in the root. Then, you start building like normal.

Now, take your big idea, spin it up quickly in micro-symfony, and build something amazing.

Seeya next time!