The new Awesome of Symfony 3.0

With <3 from SymfonyCasts

Chapter 1: Shiny new Directory Structure

Well hey guys! Symfony 3 is around the corner, ah, or maybe even released if you're watching this later! Hello from the past!

Anyways, it's time to talk about all the new cool shiny stuff. But first, I have a confession: Symfony 3 doesn't have *any* new features. Not even 1... and this is amazing. Seriously, Symfony is paving the way towards a new paradigm of upgrading. I'll talk about upgrading later. But in a nut shell, it means that you'll be able to upgrade to Symfony 3 without rewriting your code and breaking everything.

Introducing, Symfony 2.8!

When Symfony 3 is released, Symfony 2.8 will be released at the same time and both versions will have the same features. That means that if you upgrade to 2.8, you'll get all the good new stuff. So what's new in Symfony 3? Nothing! But there *are* a lot of new great features in 2.8, and I'll tell you about them.

The New Directory Structure

But wait! There *is* one thing that's new *specifically* in Symfony 3: the new Symfony 3 directory structure. You can see it in my editor. If you're an experienced user, it's not a big change.

var/

First, the app/cache and app/logs directories have been moved to a new var/ directory. The bootstrap.php.cache file also got moved here from app/.

bin/console

Second, we all know and love app/console. Well, that's now bin/console:

```
$ bin/console
```

As a result, the app/ directory is now *just* configuration code: stuff that we can actually modify. All the things we don't normally need to worry about editing - like the cache and console file - are out of the way. This gives app/ a new focus.

PhpStorm Symfony Plugin

If you're using PhpStorm with the Symfony plugin, you'll need to change a couple of settings so that it knows to look in the var/ directory for a few cache files. You may not need to do this in the future - but double-check it.

How can I Upgrade to the New Structure

So, how can we upgrade *our* project from the old structure to this one? And do we *need* to change our project?

Actually, no: this new structure has *always* been possible. In fact, open up AppKernel. The *only* reason the new structure works is because the getCacheDir() and getLogDir() methods have been overridden:

52 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
... lines 8 - 36
    public function getCacheDir()
    {
        return dirname(__DIR__).'/var/cache/'.$this->environment;
    }

    public function getLogDir()
    {
        return dirname(__DIR__).'/var/logs';
    }
... lines 46 - 50
}
```

If you don't override these, they default to using cache/ and logs/ in *this* directory: app/.

So in your project, if you override these then... boom! You're using the new directory structure. Well, you'll also want to move the console file and a few other things. I'll talk about upgrading later.

And if you *don't* want to use the new directory structure... then boom! Don't use it. You're free to organize things however you want. Just be aware that the Symfony documentation will be reference things like the bin/console and var/cache and you'll need to translate to what that means in your app.

New Tests Paths

There are two other tiny changes: the tests directory has been moved out of AppBundle into the root of the project:

19 lines | tests/AppBundle/Controller/DefaultControllerTest.php

```
... lines 1 - 2
namespace Tests\AppBundle\Controller;
... lines 4 - 6
class DefaultControllerTest extends WebTestCase
{
... lines 9 - 17
}
```

And phpunit.xml.dist also now lives in the root directory. That means no more phpunit -c app: just run phpunit:

```
$ phpunit
```

and it'll find the configuration file automatically.

That's it for the new directory structure, you can start using it right now, or leave your project alone.

Chapter 2: Guard: Joyful Authentication

My favorite new feature for Symfony 2.8 is Guard. It makes creating custom and crazy authentication systems really really easy. I'm a bit biased: Guard was my creation, inspired by a lot of people and projects.

The Weirdest Login Form Ever

Later, I'll do some in-depth screencasts about Guard, but I want to give you a taste of what's possible. In this project, I have a tradition login system:

42 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 21
    /**
     * @Route("/login", name="login")
     */
    public function sillyLoginAction()
    {
        $error = $this->get('security.authentication_utils')
            ->getLastAuthenticationError();

        return $this->render('default/login.html.twig', [
            'error' => $error
        ]);
    }
... lines 34 - 42
```

The /login controller renders a login template and this template builds a form that POSTs right back to that same loginAction:

32 lines | app/Resources/views/default/login.html.twig

```
... lines 1 - 9
    <form action="{{ path('login') }}" method="POST">
... line 11
            <label for="exampleInputEmail1">Username</label>
            <input type="text" name="_username" class="form-control" id="exampleInputEmail1" placeholder="Username">
... lines 14 - 15
            <label for="exampleInputPassword1">Password</label>
            <input type="password" name="_password" class="form-control"
                   id="exampleInputPassword1" placeholder="Password">
... lines 19 - 20
            <label for="the-answer">The answer to life the universe and everything</label>
            <input type="text" name="the_answer" class="form-control" id="the-answer">
... lines 23 - 25
                <input type="checkbox" name="terms"> I agree to want to be logged in
... lines 27 - 28
        <button type="submit" class="btn btn-default">Login</button>
    </form>
... lines 31 - 32
```

It has a username field, a password field and then a couple of extra fields. One of them is "the answer to life the universe and everything", which of course must be set to 42. The user also needs to check this "agreement" checkbox.

So how can we make a login system that checks *four* fields instead of just the usual username and password? Hello Guard.

Before coding, start the built-in web server, which is now bin/console server:run:

```
$ bin/console server:run
```

Try out /login.

Creating the Authenticator Class

To use guard, we need a new class. I'll create a new directory called Security, but that's not important. Call the class WeirdFormAuthenticator.Next, make this class implement GuardAuthenticatorInterface or extend the slightly easier AbstractGuardAuthenticator:

44 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 2
namespace AppBundle\Security;
... lines 4 - 10
use Symfony\Component\Security\Guard\AbstractGuardAuthenticator;

class WeirdFormAuthenticator extends AbstractGuardAuthenticator
{
... lines 15 - 42
}
```

In PhpStorm, I'll open the generate menu and select "Implement Methods". Select *all* of them, including start(), which is hiding at the bottom. Also move the start() method to the bottom of the class: it'll fit more thematically down there:

44 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 4
use Symfony\Component\HttpFoundation\Request;
... line 6
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
... lines 11 - 12
class WeirdFormAuthenticator extends AbstractGuardAuthenticator
{
    public function getCredentials(Request $request)
    {
    }

    public function getUser($credentials, UserProviderInterface $userProvider)
    {
    }

    public function checkCredentials($credentials, UserInterface $user)
    {
    }

    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
    }

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
    {
    }

    public function supportsRememberMe()
    {
    }

    public function start(Request $request, AuthenticationException $authException = null)
    {

    }
}
```

Your job is simple: fill in each of these methods.

getCredentials()

Once we hook it up, the getCredentials() method will be called on the authenticator on *every single request*. This is your opportunity to collect the username, password, API token or whatever "credentials" the user is sending in the request.

For this system, we don't need to bother looking for the credentials unless the user is submitting the login form. Add an if statement: if $request->getPathInfo() - that's the current, cleaned URL - != /login or !$request->isMethod('POST'), then return null:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 28
    public function getCredentials(Request $request)
    {
        if ($request->getPathInfo() != '/login' || !$request->isMethod('POST')) {
            return;
        }
... lines 34 - 40
    }
... lines 42 - 101
```

When you return null from getCredentials(), no other methods will be called on the authenticator.

Below that, return an array that contains any credential information we need. Add a username key set to $request->request->get('_username'): that's the name of the field in the form:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 28
    public function getCredentials(Request $request)
    {
... lines 31 - 34
        return [
            'username' => $request->request->get('_username'),
... lines 37 - 39
        ];
    }
... lines 42 - 101
```

Repeat that for password and answer. In the form, answer's name is the_answer and the last is named terms. Finish the array by fetching the_answer and terms:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 28
    public function getCredentials(Request $request)
    {
... lines 31 - 34
        return [
            'username' => $request->request->get('_username'),
            'password' => $request->request->get('_password'),
            'answer'   => $request->request->get('the_answer'),
            'terms'    => $request->request->get('terms'),
        ];
    }
... lines 42 - 101
```

The keys on the right are obviously the names of the submitted form fields. The keys on the left can be anything: you'll see how to use them soon.

getUser()

If getCredentials() returns a non-null value, then getUser() is called next. I have a really simple User entity that has basically just one field: username:

56 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 11
class User implements UserInterface
{
... lines 14 - 20
    /**
     * @ORM\Column(type="string")
     */
    private $username;
... lines 25 - 54
}
```

I'm not even storing a password. Whatever your situation, just make sure you have a User class that implements UserInterface *and* a "user provider" for it that's configured in security.yml:

31 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 5
    providers:
        my_entity_users:
            entity:
                class: AppBundle:User
... lines 10 - 31
```

See the $credentials variable? That's equal to whatever you returned from getCredentials(). Set a new $username variable by grabbing the username key from it:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 17
class WeirdFormAuthenticator extends AbstractGuardAuthenticator
{
... lines 20 - 42
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $username = $credentials['username'];
... lines 46 - 53
    }
... lines 55 - 99
}
```

Let's get crazy and say: "Hey, if you have a username that starts with an @ symbol, that's not okay and you can't login". To fail authentication, return null:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 42
43      public function getUser($credentials, UserProviderInterface $userProvider)
44      {
45          $username = $credentials['username'];
46
47          // this looks like a weird username
48          if (substr($username, 0, 1) == '@') {
49              return;
50          }
... lines 51 - 53
54      }
... lines 55 - 101
```

Now, query for the User! We need the entity manager, so add a private $em property and generate the constructor. Type-hint it with EntityManager:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 5
6   use Doctrine\ORM\EntityManager;
... lines 7 - 17
18  class WeirdFormAuthenticator extends AbstractGuardAuthenticator
19  {
20      private $em;
... lines 21 - 22
23      public function __construct(EntityManager $em, RouterInterface $router)
24      {
25          $this->em = $em;
... line 26
27      }
... lines 28 - 99
100 }
```

Back in getUser(), this is easy: return $this->em->getRepository('AppBundle:User')->findOneBy() with username equals $username:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 42
43      public function getUser($credentials, UserProviderInterface $userProvider)
44      {
45          $username = $credentials['username'];
... lines 46 - 51
52          return $this->em->getRepository('AppBundle:User')
53              ->findOneBy(['username' => $username]);
54      }
... lines 55 - 101
```

The job of getUser() is to return a User object or null to fail authentication. It's perfect.

checkCredentials()

If getUser() *does* return a User, checkCredentials() is called. This is where you check if the password is valid or anything else to determine that the authentication request is legitimate.

And surprise! The $credentials argument is once again what you returned from getCredentials().

Let's check a few things. First, the user doesn't have a password stored in the database. In my system, the password for everybody is the same: symfony3. If it doesn't equal symfony3, return null and authentication will fail:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 55
    public function checkCredentials($credentials, UserInterface $user)
    {
        if ($credentials['password'] != 'symfony3') {
            return;
        }
... lines 61 - 70
    }
... lines 72 - 101
```

Second, if the answer does not equal 42, return null. And finally, if the terms weren't checked, you guessed it, return null. Authentication will fail unless checkCredentials() exactly returns true. So return that at the bottom:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 55
    public function checkCredentials($credentials, UserInterface $user)
    {
... lines 58 - 61
        if ($credentials['answer'] != 42) {
            return;
        }

        if (!$credentials['terms']) {
            return;
        }

        return true;
    }
... lines 72 - 101
```

Handling Authentication Success and Failure

That's it! The last few methods handle what happens on authentication failure and success. Both should return a Response object, or null to do nothing and let the request continue.

onAuthenticationFailure()

The $exception passed to onAuthenticationFailure holds details about *what* went wrong:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 72
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
... lines 75 - 79
    }
... lines 81 - 101
```

Was the user not found? Were the credentials wrong? Store this in the session so we can show it to the user. For the key, use the constant: Security::AUTHENTICATION_ERROR:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 72
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        $request->getSession()->set(Security::AUTHENTICATION_ERROR, $exception);
... lines 76 - 79
    }
... lines 81 - 101
```

This is *exactly* what the normal form login system does: it adds an error to this same key on the session.

In the controller, the security.authentication.utils service reads this key from the session when you call getLastAuthenticationError():

42 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 8
class DefaultController extends Controller
{
... lines 11 - 21
    /**
     * @Route("/login", name="login")
     */
    public function sillyLoginAction()
    {
        $error = $this->get('security.authentication_utils')
            ->getLastAuthenticationError();
... lines 29 - 32
    }
... lines 34 - 40
}
```

Other than storing the error, what we *really* want to do when authentication fails is redirect back to the login page. To do this, add the Router as an argument to the constructor and use that to set a new property. I'll do that with a shortcut:

101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 9
use Symfony\Component\Routing\RouterInterface;
... lines 11 - 17
class WeirdFormAuthenticator extends AbstractGuardAuthenticator
{
... line 20
    private $router;

    public function __construct(EntityManager $em, RouterInterface $router)
    {
... line 25
        $this->router = $router;
    }
... lines 28 - 99
}
```

Now it's simple: $url = $this->router->generate() and pass it login - that's the route name to my login page. Then, return a new RedirectResponse:

```
101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php
    ... lines 1 - 72
73      public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
74      {
    ... lines 75 - 76
77          $url = $this->router->generate('login');
78
79          return new RedirectResponse($url);
80      }
    ... lines 81 - 101
```

onAuthenticationSuccess()

When authentication works, keep it simple and redirect back to the homepage. In a real app, you'll want to redirect them back to the previous page:

```
101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php
    ... lines 1 - 81
82      public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
83      {
84          $url = $this->router->generate('homepage');
85
86          return new RedirectResponse($url);
87      }
    ... lines 88 - 101
```

There's a base class called AbstractFormLoginAuthenticator that can help with this.

start() - Inviting the User to Authentication

The start() method is called when the user tries to access a page that requires login as an anonymous user. For this situation, redirect them to the login page:

```
101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php
    ... lines 1 - 88
89      public function start(Request $request, AuthenticationException $authException = null)
90      {
91          $url = $this->router->generate('login');
92
93          return new RedirectResponse($url);
94      }
    ... lines 95 - 101
```

supportsRememberMe()

Finally, if you want to be able to support remember me functionality, return true from supportsRememberMe():

```
101 lines | src/AppBundle/Security/WeirdFormAuthenticator.php
    ... lines 1 - 95
96      public function supportsRememberMe()
97      {
98          return true;
99      }
    ... lines 100 - 101
```

You'll still need to configure the remember_me key in the firewall.

Configuring the Authenticator

That's it! Now we need to tell Symfony about the authentication with two steps. The first shouldn't surprise you: register this as a service. In services.yml create a new service - how about weird_authenticator. The class is WeirdFormLoginAuthenticator and there are two arguments: the entity manager and the router:

```
10 lines | app/config/services.yml
... lines 1 - 5
services:
    weird_authenticator:
        class: AppBundle\Security\WeirdFormAuthenticator
        arguments: ['@doctrine.orm.entity_manager', '@router']
```

Finally, hook this up in security.yml. Under your firewall, add a guard key and an authenticators key below that. Add one authenticator - weird_authenticator - the service name:

```
31 lines | app/config/security.yml
... lines 1 - 2
security:
... lines 4 - 10
    firewalls:
... lines 12 - 16
        main:
            anonymous: ~
... lines 19 - 21
            guard:
                authenticators:
                    - weird_authenticator
... lines 25 - 31
```

Now, getCredentials() will be called on every request. If it returns something other than null, getUser() will be called. If this returns a User, then checkCredentials() will be called. And if this returns true, authentication will pass.

Test the Login

Try it out! With a blank form, it says Username could not be found. It *is* looking for the credentials on the request, but it fails on the getUser() method. I do have one user in the database: weaverryan. Put symfony3 for the password but set the answer to 50. This fails with "Invalid Credential". Finally, fill in everything correctly. And it works! The web debug toolbar congratulates us: logged in as weaverryan.

Customize the Error Messages

We saw two different error messages, depending on whether authentication failed in getUser() or checkCredentials(). But you can *completely* control these messages by throwing a CustomUserMessageAuthenticationException(). I know, it has a long name, but it's job is clear: pass it the message you want want to show the user, like:

108 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 12
use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
... lines 14 - 43
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
... lines 46 - 48
        if (substr($username, 0, 1) == '@') {
            throw new CustomUserMessageAuthenticationException(
                'Starting a username with @ is weird, don\'t you think?'
            );
        }
... lines 54 - 56
    }
... lines 58 - 108
```

Below if the answer is wrong, do the same thing: throw new CustomUserMessageAuthenticationException() with:

108 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 58
    public function checkCredentials($credentials, UserInterface $user)
    {
... lines 61 - 64
        if ($credentials['answer'] != 42) {
            throw new CustomUserMessageAuthenticationException(
                'Don\'t you read any books?'
            );
        }
... lines 70 - 77
    }
... lines 79 - 108
```

And in case the terms checkbox isn't checked, throw a new CustomUserMessageAuthenticationException and threaten them to agree to our terms!

108 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 58
    public function checkCredentials($credentials, UserInterface $user)
    {
... lines 61 - 70
        if (!$credentials['terms']) {
            throw new CustomUserMessageAuthenticationException(
                'Agree to our terms!!!'
            );
        }

        return true;
    }
... lines 79 - 108
```

Try it out with @weaverryan. There's the first message! Try 50 for the answer: there's the second message. And if you fill in everything right but have no checkbox, you see the final message.

That's Guard authentication: create one class and do whatever the heck that you want. I showed a login form, but it's even easier to use for API authentication. Basically, authentication is not hard anymore. Invent whatever insane system you want and give the user exactly the message you want. If you want to return JSON on authentication failure and success instead of redirecting, awesome, do it.

Chapter 3: Micro Symfony via MicroKernelTrait

How big is Symfony, really? It's HUGE! I'm kidding. Dude, Symfony is just a bunch of little libraries, so we can make it as small or as huge as we want. Most of us start with the *Standard Edition*: it looks basically like this, 10 or so configuration files, two front controllers, and some other skeleton files. Probably 20 or 30 files to begin with.

But no more in Symfony 2.8! Ok, the Standard Edition of course still exists, but now we have a brand new tool in the arsenal: the MicroKernelTrait. You can build a Symfony app that's as small as one file, then let it grow from there.

Hey, should we try it?

Basic "Micro" Kernel

Imagine that we don't have this big project: we *only* have a composer.json file that requires symfony/symfony:

64 lines | composer.json

```
{
... lines 2 - 12
    "require": {
... line 14
        "symfony/symfony": "3.0.*@dev",
... lines 16 - 25
    },
... lines 27 - 62
}
```

In the app directory, create a new class called LittleKernel. The kernel is the heart of your application: make it extend the base Kernel class:

8 lines | app/LittleKernel.php

```
... lines 1 - 2
use Symfony\Component\HttpKernel\Kernel;

class LittleKernel extends Kernel
{
}
```

Normally, this class *must* implement two abstract methods: registerBundles() and registerContainerConfiguration(). But let's *not* implement those. Instead, use the new MicroKernelTrait:

26 lines | app/LittleKernel.php

```
... lines 1 - 2
use Symfony\Bundle\FrameworkBundle\Kernel\MicroKernelTrait;
... lines 4 - 9
class LittleKernel extends Kernel
{
    use MicroKernelTrait;
... lines 13 - 24
}
```

This implements registerContainerConfiguration() *for* us. All we need to do is add registerBundles(), configureRoutes() and configureContainer():

26 lines | app/LittleKernel.php

```
... lines 1 - 2
use Symfony\Bundle\FrameworkBundle\Kernel\MicroKernelTrait;
use Symfony\Component\Config\Loader\LoaderInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\HttpKernel\Bundle\BundleInterface;
use Symfony\Component\HttpKernel\Kernel;
use Symfony\Component\Routing\RouteCollectionBuilder;

class LittleKernel extends Kernel
{
    use MicroKernelTrait;

    public function registerBundles()
    {
    }

    protected function configureRoutes(RouteCollectionBuilder $routes)
    {
    }

    protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
    {
    }
}
```

Ok, stop! Check this out, it's *really* cool. What *is* an application? Well, it's a set of bundles, a collection of services, and a list of routes. And would you look at that! We have exactly three methods that define the *three* things that make up a framework.

Single-File Symfony

Let's make a real-live app in just this one file. First, register some bundles:

40 lines | app/LittleKernel.php

```
... lines 1 - 2
use Symfony\Bundle\FrameworkBundle\FrameworkBundle;
... lines 4 - 11
class LittleKernel extends Kernel
{
... lines 14 - 15
    public function registerBundles()
    {
        return [
            new FrameworkBundle()
        ];
    }
... lines 22 - 38
}
```

We only need *one* bundle: FrameworkBundle. To configure that bundle, head to configureContainer(), call $c->loadFromExtension(), and pass it framework and an array of configuration. The only key it *needs* is secret. Set it to the top secret super cool-kids password: micr0:

40 lines | app/LittleKernel.php

```
... lines 1 - 32
    protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
    {
        $c->loadFromExtension('framework', [
            'secret' => 'micr0',
        ]);
    }
... lines 39 - 40
```

And guess what? We already know *exactly* how this loadFromExtension() works: it's the PHP equivalent of having a config.yml file with a framework key at the root and a secret key below that:

61 lines | app/config/config.yml

```
... lines 1 - 10
framework:
... lines 12 - 13
    secret:        "%secret%"
... lines 15 - 61
```

Oh, and yea, you *can* still load YML and XML config files, and you probably will. I'll mention that soon.

At this point, this app will *already* work. But let's make a page. In configureRoutes(), we receive a shiny new RouteCollectionBuilder object - *another* new thing for Symfony 2.8 that makes adding routes in PHP a lot more fun than it used to be.

Add a route with $routes->add('/hello/symfony/{version}'). The second argument is the controller: pass it kernel:helloSymfony:

40 lines | app/LittleKernel.php

```
... lines 1 - 27
    protected function configureRoutes(RouteCollectionBuilder $routes)
    {
        $routes->add('/hello/symfony/{version}', 'kernel:helloSymfony');
    }
... lines 32 - 40
```

This is the *service* format for configuring a controller: kernel is the name of the service - that actually points to *this* object - and helloSymfony is the name of the method.

Hey, let's build that: public function helloSymfony(). Give it a $version argument and say return new Response('Hi Symfony version '.$version);:

40 lines | app/LittleKernel.php

```
... lines 1 - 22
    public function helloSymfony($version)
    {
        return new Response('Hi Symfony version '.$version);
    }
... lines 27 - 40
```

OMG. That's a fully-functional Symfony app in one file. The only thing we're missing is a front controller that boots and runs this guy. To be *really* trendy, we could put that code at the bottom of this file. But come on guys - I think we're letting this micro-framework fad get to our heads. *Some* structure is a good thing.

Adding the Front Controller

Instead, in the web directory, create a new file called tiny.php. This will be our version of an app.php or app_dev.php file.

Start by requiring Composer's autoloader and our LittleKernel.php, since it isn't configured to be autoloaded:

13 lines | web/tiny.php

```
... lines 1 - 4
require __DIR__.'/../vendor/autoload.php';
require __DIR__.'/../app/LittleKernel.php';
... lines 7 - 13
```

The rest of this file is just the same boring stuff we see in app_dev.php. In fact, copy the bottom of that file and paste it here. Change AppKernel to LittleKernel and make sure the Request class has its use statement. Remove loadClassCache() - a small performance line - to make this *even* smaller:

13 lines | web/tiny.php

```
... lines 1 - 2
use Symfony\Component\HttpFoundation\Request;
... lines 4 - 7
$kernel = new LittleKernel('dev', true);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

We're crazy!

Ready? Try it!

Over in the browser add /tiny.php to the URL:

> http://localhost:8000/tiny.php

Ah! Hide from the error! Wait, check it out, it says:

> No route found for "GET /" in...

Our app is working! We're using *such* a small bit of Symfony that we don't even have the nice exception pages. You can get these back by adding TwigBundle, but it's not *technically* necessary.

Try the real page now: /hello/symfony/3. There it is!

Creating a Bigger (Micro) App

A real app won't be *just* one file. But the MicroKernelTrait allows you to grow and opt into whatever features you want. Let's see if we can get our existing app running. That means loading annotation routes from DefaultController and booting Twig to render the templates.

In LittleKernel, add TwigBundle and SensioFrameworkExtraBundle to registerBundles():

48 lines | app/LittleKernel.php

```
... lines 1 - 2
use Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle;
... lines 4 - 5
use Symfony\Bundle\TwigBundle\TwigBundle;
... lines 7 - 13
class LittleKernel extends Kernel
{
... lines 16 - 17
    public function registerBundles()
    {
        return [
            new FrameworkBundle(),
            new TwigBundle(),
            new SensioFrameworkExtraBundle()
        ];
    }
... lines 26 - 46
}
```

To *activate* Twig, we need more configuration under the framework key. Add a templating key with an engines sub-key. In there, add twig in an array:

48 lines | app/LittleKernel.php

```
... lines 1 - 38
    protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
    {
        $c->loadFromExtension('framework', [
            'secret' => 'micr0',
            'templating' => ['engines' => ['twig']],
... line 44
        ]);
    }
... lines 47 - 48
```

That configuration looks a little "deep", but it's *exactly* what you've always had in your config.yml file:

61 lines | app/config/config.yml

```
... lines 1 - 10
framework:
... lines 12 - 21
    templating:
        engines: ['twig']
... lines 24 - 61
```

Also activate the assets subsystem with an assets key set to an empty array. This makes it possible to use the asset() function in Twig:

48 lines | app/LittleKernel.php

```
... lines 1 - 38
    protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
    {
        $c->loadFromExtension('framework', [
            'secret' => 'micr0',
            'templating' => ['engines' => ['twig']],
            'assets' => []
        ]);
    }
... lines 47 - 48
```

Loading External Routing Files

Next, we need to load the annotation routes. That's *super* easy.

Use $routes->import(). Pass it whatever routing resource you want, like a YAML or XML file. In our case, import the annotations from the Controller directory with __DIR__.'/../src/AppBundle/Controller'. Pass / as the second argument - that's the prefix - and annotation - the "type" - as the last arg:

48 lines | app/LittleKernel.php

```
... lines 1 - 31
    protected function configureRoutes(RouteCollectionBuilder $routes)
    {
... lines 34 - 35
        $routes->import(__DIR__.'/../src/AppBundle/Controller', '/', 'annotation');
    }
... lines 38 - 48
```

This should feel familiar too: it's equivalent to importing routes in a YAML file with resource and type keys:

4 lines | app/config/routing.yml

```
app:
    resource: "@AppBundle/Controller/"
    type:     annotation
```

Really it's the exact same thing, just done in PHP instead.

We're done! But please, please don't get too excited, it's not going to work quite yet. Refresh anyways to see what's happening.

Annotations Loader

Error!

> The annotation '@Sensio\Bundle\FrameworkExtraBundle\Configuration\Route' in method could not be auto-loaded.

If you work with annotations, then you need one little extra line of code. This line already exists inside of Symfony's normal app/autoload.php file:

16 lines | app/autoload.php

```
... lines 1 - 2
use Doctrine\Common\Annotations\AnnotationRegistry;
... lines 4 - 12
AnnotationRegistry::registerLoader(array($loader, 'loadClass'));
... lines 14 - 16
```

In tiny.php, require *this* autoload file. That gives us the annotation line we need *and* still initializes Composer's autoloader:

13 lines | web/tiny.php

```
... lines 1 - 4
require __DIR__.'/../app/autoload.php';
... lines 6 - 13
```

Refresh! It's alive!

Opting Into Features

Now check out the homepage. It's dead again!

> Unknown "is_granted" function in base.html.twig at line 19.

Open that file and find line 19. Our app is choking on is_granted():

39 lines | app/Resources/views/base.html.twig

```
... lines 1 - 16
            <ul class="nav nav-pills pull-right">
                <li role="presentation" class="active"><a href="{{ path('homepage') }}">Home</a></li>
                {% if is_granted('IS_AUTHENTICATED_FULLY') %}
                    <li role="presentation"><a href="{{ path('logout') }}">Logout</a></li>
                {% else %}
                    <li role="presentation"><a href="{{ path('login') }}">Login</a></li>
                {% endif %}
            </ul>
... lines 25 - 39
```

This is one of the best and worst parts of the MicroKernel. If you're accustomed to having twenty bundles, then you're accustomed to having *all* the features of Symfony. But, with the MicroKernel you must opt into each feature. The is_granted() comes from Symfony's SecurityBundle... and we're not using that! If we need it, then you'll need to add it to LittleKernel and configure your security.yml file.

For now, remove is_granted():

35 lines | app/Resources/views/base.html.twig

```
... lines 1 - 16
            <ul class="nav nav-pills pull-right">
                <li role="presentation" class="active"><a href="{{ path('homepage') }}">Home</a></li>

            </ul>
... lines 21 - 35
```

And try again.

It alive... again! That's the MicroKernelTrait in a nut shell: add as many bundles as you want, configure them and add routing. You *can* still load external routing and configuration files, and you should! The point isn't to put *everything* into a single file: it's about starting with a single file and then choosing which bundles and configurations make sense for your project.

Right now, I have my framework configuration in PHP. But if the project grows, I might want to use a YAML file instead. That's no problem.

48 lines | app/LittleKernel.php

```
... lines 1 - 38
    protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
... lines 40 - 48
```

See that $loader variable? It has an import() method on it - use that to import a config.yml file and you're done. This is the PHP-equivalent to the imports line in YAML.

Go play with it: I hope you love it as much as I do!

Chapter 4: New Profiler

Change the URL back to our *real*, non-micro app:

> http://localhost:8000

We see the same page, but with the sweet black web debug toolbar. This is the same toolbar that you've always known and loved, but redesigned and rethought. That means it now has a sleek black interface with flat UI elements, because they're super hip!

There's also a lot less stuff on it. Don't worry, it *is* still collecting all the same stuff as before, but only the important details go here, like the template rendering time, who's logged in, and how long the page took to load. This is *much* better than 2.7, where my toolbar was breaking onto multiple lines.

Just like always, you can click on any icon to arrive at the web profiler. This is also completely redesigned to be simpler to use and easier on the eyes.

Profiling API/AJAX Requests

Click that Last 10 link. This is *not* a new feature, but it's totally underused! This lists all the previously profiled URLs. So for example, head to http://localhost:8000/login and refresh this page. Hello /login! If we clicked the 477... token URL, we'd see all the profiler info from that request.

Why do I love this? It's super handy when you're making API calls or Ajax requests. If something goes wrong and you want to look into, open a tab, go to /_profiler and find that request at the top of this list.

Check it out: go to your editor and open up DefaultController. Go down to sillyLoginAction(). OK, pretend that something went wrong and we can't figure it out. If this security.authentication_utils is the problem, we might want to use dump() to print it out. Below that, throw a new Exception to simulate something going wrong with the helpful message of "Something went wrong!":

46 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 2
3   namespace AppBundle\Controller;
... lines 4 - 8
9   class DefaultController extends Controller
10  {
... lines 11 - 21
22      /**
23       * @Route("/login", name="login")
24       */
25      public function sillyLoginAction()
26      {
27          $error = $this->get('security.authentication_utils')
28              ->getLastAuthenticationError();
29
30          dump($this->get('security.authentication_utils'));
31
32          throw new \Exception('Something went wrong!');
33
34          return $this->render('default/login.html.twig', [
35              'error' => $error
36          ]);
37      }
... lines 38 - 44
45  }
```

Reload /login. There's the error! And there's the dumped variable down in the toolbar. But what if this wasn't a big beautiful HTML web page but some API endpoint you're testing!? Copy the URL, head to the terminal and curl the URL:

```
$ curl http://localhost:8000/login
```

Ah, gross, disgusting, horrible - HTML in the terminal! Unless you *love* reading raw HTML, it's tough to see what went wrong. And seeing the dumped variable... well... that's impossible!

Go back to /_profiler. There's our shiny 500 error. Click to get the details. This is awesome for two reasons. First, down in the Debug tab, you can see the dumped variable. Woot! You *can* dump a variable in your API and then see it here.

Second, you can see the big beautiful HTML exception page as if you had looked at it in a browser. Hello sweet debugging.

So don't be crazy! Use dump() and the profiler for your API: it's one of the useful tools that everyone loves to forget about.

This feature isn't new in 2.8, but this fancy new look is.

Chapter 5: Service Autowiring

The next new feature is called autowiring. And yes, it *is* magic. And yes, it's super controversial - it's like the celebrity gossip of the Symfony world, right along side annotations.

Autowiring makes registering services easier. Normally, a service needs a name, a class *and* its constructor arguments:

10 lines | app/config/services.yml

```
... lines 1 - 5
services:
    weird_authenticator:
        class: AppBundle\Security\WeirdFormAuthenticator
        arguments: ['@doctrine.orm.entity_manager', '@router']
```

What autowiring says is, "maybe we don't need to give it the arguments?". So remove the arguments key and instead type autowire: true:

11 lines | app/config/services.yml

```
... lines 1 - 5
services:
    weird_authenticator:
        class: AppBundle\Security\WeirdFormAuthenticator
        # arguments: ['@doctrine.orm.entity_manager', '@router']
        autowire: true
```

Head back to the browser, refresh the login page and press the login button. It doesn't explode! How was it able to create the WeirdFormAuthenticator!!!??? That's autowiring.

How does it Work?

It works via type-hints. In WeirdFormAuthenticator we're type-hinting the first argument with EntityManager and the second argument with RouterInterface:

108 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 2
namespace AppBundle\Security;
... lines 4 - 18
class WeirdFormAuthenticator extends AbstractGuardAuthenticator
{
... lines 21 - 23
    public function __construct(EntityManager $em, RouterInterface $router)
    {
... lines 26 - 27
    }
... lines 29 - 106
}
```

Behind the scenes, well, in a "compiler pass" if you're super geeky and curious, autowiring looks in the container and says "could you please show me all the services type-hinted with EntityManager?". Since there is only one, it auto-wires that service as the first argument.

The same is true for RouterInterface: it sees that there is only one service whose class implements RouterInterface, so it injects the router service.

What if there are Multiple Services with the Class?

But what if there are *multiple* services that implement an interface or match a class? Well, frankly, your computer will catch on fire....

I'm kidding! Autowiring won't work, but it *will* throw a clear exception. Autowiring is magical... but not completely *magical*: it won't try to guess *which* entity manager you want. It's the Symfony-spin on auto-wiring.

If you do hit this problem, there is a way for you to tell the container *which* service you want to inject for the type-hinted class.

But the real beauty of autowiring is when it just works. The *whole* point is to save time so you can rapidly develop.

Moar Magic: Injecting Non-Existent Services

Autowiring has one more interesting trick. In the Security directory, create a new super important class called EvilSecurityRobot. Give it one method: public function doesRobotAllowAccess():

12 lines | src/AppBundle/Security/EvilSecurityRobot.php

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

class EvilSecurityRobot
{
    public function doesRobotAllowAccess()
    {
        return rand(0, 10) >= 5;
    }
}
```

Basically, the evil robot will decide, randomly, whether or not we can login. So even if I enter *all* the login fields correctly, the evil security robot could still say "NOPE! You're not getting in and I'm not sorry. <3 The Evil Robot".

The EvilSecurityRobot is ready. To use this in WeirdFormAuthenticator, pass it as the third argument to the constructor: EvilSecurityRobot $robot. Now create a $robot property and set it:

116 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 18
class WeirdFormAuthenticator extends AbstractGuardAuthenticator
{
... lines 21 - 22
    private $robot;
... line 24
    public function __construct(EntityManager $em, RouterInterface $router, EvilSecurityRobot $robot)
    {
... lines 27 - 28
        $this->robot = $robot;
    }
... lines 31 - 114
}
```

In checkCredentials(), if (!$this->robot->doesRobotAllowAccess()) then throw a really clear new CustomUserMessageAuthenticationException() that says "RANDOM SECURITY ROBOT SAYS NO!":

116 lines | src/AppBundle/Security/WeirdFormAuthenticator.php

```
... lines 1 - 60
61      public function checkCredentials($credentials, UserInterface $user)
62      {
63          if (!$this->robot->doesRobotAllowAccess()) {
64              throw new CustomUserMessageAuthenticationException(
65                  'RANDOM SECURITY ROBOT SAYS NO!'
66              );
67          }
... lines 68 - 85
86      }
... lines 87 - 116
```

And I'll even put quotes around that.

This is when we would *normally* go to services.yml and update the arguments key to inject the EvilSecurityRobot. But wait! Before we do that, we need to register the EvilSecurityRobot as a service first.

Resist the urge! Instead, do nothing: absolutely nothing. Refresh the page. Fill in weaverryan and try to login until you se the error. Bam!

> RANDOM SECURITY ROBOT SAYS NO!

It works! What the heck is going on?

Remember, weird_authenticator is autowired. Because of that, Symfony sees that the third argument is type-hinted with EvilSecurityRobot. Next, it looks in the container, but finds *no* services with this class. But instead of failing, it creates a *private* service in the container automatically and injects that. This actually works pretty well: EvilSecurityRobot itself is created with autowiring. So if *it* had a couple of constructor arguments, it would try to autowire those automatically.

Oh, and if you have multiple autowired services that need an EvilSecurityRobot, the container will create just *one* private service and re-use it.

Some people will *love* this new feature and some people will *hate* it and complain on Twitter. But it's cool: like most things, you're not forced to use it. But, if you're doing some rapid application development, try it! It won't work in all cases, and isn't able to inject configuration yet, but when it works, it can save time, just like it did here for us.

Wield this new weapon wisely.

Chapter 6: Form Updates

I know, I know, everyone *loves* it when the form system changes and breaks everything. Well, get ready: there are more changes in Symfony 2.8. And at first, they're a little shocking. But I think you'll like them: it's a debateable step towards simplification.

Let's create a registration form using this mysterious *new* stuff. Add a Form directory and create a RegistrationForm class inside. PhpStorm gives me a nice skeleton for the class.... *but* now it has too much. Remove the getName() method at the bottom:

21 lines | src/AppBundle/Form/RegistrationForm.php

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

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class RegistrationForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {

    }

    public function configureOptions(OptionsResolver $resolver)
    {

    }
}
```

This was always a useless, but required function. But no more!

For the most part, forms look the same. In configureOptions() call $resolver->setDefaults() and pass it data_class set to AppBundle\Entity\User:

24 lines | src/AppBundle/Form/RegistrationForm.php

```
... lines 1 - 16
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => 'AppBundle\Entity\User'
        ]);
    }
... lines 23 - 24
```

My super-simple user has just *one* field on it: username, which makes this one of the most ridiculously useless registration forms in history:

56 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 11
class User implements UserInterface
{
... lines 14 - 20
    /**
     * @ORM\Column(type="string")
     */
    private $username;
... lines 25 - 30
    public function getUsername()
    {
        return $this->username;
    }
... lines 35 - 50
    public function setUsername($username)
    {
        $this->username = $username;
    }
}
```

Form types as Class Names!

Add our one field with $builder->add('username'). But stop! Here is where things are different. Before I would have passed text as the second argument. But now, I'm going pass the full class name to the class that's *behind* the text field type.

To make things easier, use a shortcut: TextType::class:

24 lines | src/AppBundle/Form/RegistrationForm.php

```
... lines 1 - 5
use Symfony\Component\Form\Extension\Core\Type\TextType;
... lines 7 - 9
class RegistrationForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('username', TextType::class);
    }
... lines 16 - 22
}
```

PhpStorm doesn't auto-complete classes when you do this... yet. But if you focus on the class, a light-bulb will allow you to import the class. Or hit option+enter on a Mac to bring it up immediately. This adds the use statement. Using ::class is new in PHP 5.5... so this change won't be much fun unless you can use this. But, you can keep using the old syntax until you switch to 3.0... which requires 5.5 anyways.

Phew! That's big thing number one: instead of weird strings, we have full class names. This sucks because... well, this is a little bit more typing. But this is *sweet* because a class name is more meaningful than a string like text. If this was your first time using Symfony, you'd have a better chance of understanding how things work. And I really like that.

Create your Form: Still a Class Name

Time to use this! Open DefaultController and create a new registerAction(). Set the URL to /register call it user_register:

60 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 2
3   namespace AppBundle\Controller;
... lines 4 - 9
10  class DefaultController extends Controller
11  {
... lines 12 - 42
43      /**
44       * @Route("/register", name="user_register")
45       */
46      public function registerAction(Request $request)
47      {
... lines 48 - 57
58      }
59  }
```

Now, start just like normal with: $form = $this->createForm(). But stop! I know you *want* to say new RegistrationForm(). Instead, type RegistrationForm::class. Then, move your cursor to the class, hit option+enter and import the class:

60 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 4
5   use AppBundle\Form\RegistrationForm;
... lines 6 - 9
10  class DefaultController extends Controller
11  {
... lines 12 - 45
46      public function registerAction(Request $request)
47      {
48          $form = $this->createForm(RegistrationForm::class);
... lines 49 - 57
58      }
59  }
```

Oh and change type to class, duh! Importing the class added the use statement on top.

So wow, not only do we *not* use magic strings anymore, we also *never* pass objects. It's perfectly consistent: we *always* refer to forms and fields using the full class name. There's an important consequence: the RegistrationForm will be created and passed *no* constructor arguments.

Form Type Constructor Args?

If you need to pass something to that object, you have two options - neither of which are new. First, you can pass stuff through the $options array:

24 lines | src/AppBundle/Form/RegistrationForm.php

```
... lines 1 - 11
12      public function buildForm(FormBuilderInterface $builder, array $options)
... lines 13 - 24
```

The third argument to createForm():

60 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 47
48          $form = $this->createForm(RegistrationForm::class);
... lines 49 - 60
```

Second, you can still - like always - register your form type as a service. If RegistrationForm has constructor args, that's the real answer: register it as a service and tag it with form.type. You'll *still* refer to it via the class name, but Symfony will use

your service instead of creating a new object.

Add $request as an argument and then add the normal $form->handleRequest($request). if ($form->isValid()), for now dump($form->getData()); and put a die statement to celebrate:

60 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 9
class DefaultController extends Controller
{
... lines 12 - 45
    public function registerAction(Request $request)
    {
        $form = $this->createForm(RegistrationForm::class);
        $form->handleRequest($request);

        if ($form->isValid()) {
            dump($form->getData());die;
        }
... lines 54 - 57
    }
}
```

Finally, at the bottom, return $this->render('default/register.html.twig'). Pass it the form with 'form' => $form->createView(),:

60 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 54
        return $this->render('default/register.html.twig', [
            'form' => $form->createView()
        ]);
... lines 58 - 60
```

Simple enough and the same as always!

In app/Resources/views/default, create that register.html.twig template. I'll copy in some boilerplate code for this form:

11 lines | app/Resources/views/default/register.html.twig

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

{% block body %}
    <h1>You should signup!</h1>

    {{ form_start(form) }}
        {{ form_widget(form) }}

        <button type="submit" class="btn btn-default">Register!</button>
    {{ form_end(form) }}
{% endblock %}
```

None of this has changed. This opens the form tag, dumps out all the fields, adds a submit button and closes the form.

Time to test it! Head to /register, throw in our username: leannapelham and hit register. There's our dump!

Here's the moral of the story: forms work exactly like before. But now, form classes are *always* referred to by their class name: never objects and never the short, but weird magic string from the past.

This might be the biggest change in Symfony 3 that people will complain about, and rightly so: if you have a lot of forms, this is a lot of changes. For that, check out the Symfony Upgrade Fixer: a library that can automate some of the tasks of upgrading, including this one. Oh, and it's made by my friend Saša, and he's a really cool dude.

Chapter 7: Console Styling

We all love ice cream and Symfony's console. We love the ability to add colors, beer shaped progress bars, tables and a bunch of other cool stuff.

A little feature snuck into Symfony 2.7 and was improved greatly in Symfony 2.8. It's called the SymfonyStyle. It's more than just a cool way to dress, it's the Twitter Bootstrap for styling console commands. Ok ok, it's not a huge deal, but let's face it, that blog post with the beer icon in the console? It's pretty much our most popular blog post... so clearly you guys love this stuff.

Create a new Command directory. Inside, I'll take the lazy road and have PhpStorm make me a new command called StylesPlayCommand. Give it a name! styles:play:

22 lines | src/AppBundle/Command/StylesPlayCommand.php

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

use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class StylesPlayCommand extends ContainerAwareCommand
{
    protected function configure()
    {
        $this
            ->setName('styles:play');
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $output->writeln('boring...');
    }
}
```

Start like we always do, with lame, worn-out $output->writeln('boring'). Zip over to the terminal and try that out:

```
$ ./bin/console styles:play
```

And there it is! In all its white text on a black background glory.

Introducing SymfonyStyle

Enough of that! That still works, it will always work. But now, create a new $style variable set to new SymfonyStyle(). Pass it the $input and $output:

29 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 17
    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $style = new SymfonyStyle($input, $output);
... lines 21 - 26
    }
... lines 28 - 29
```

This class comes from a few friends of mine, Kevin Bond & Javier Eguiluz. This dynamic Canadian-Spaniard team sat down and designed a good-looking style guide that can be used for all commands in the Symfony ecosystem. As Javier put it, this is basically the stylesheet for your commands. We use simple methods, and Javier makes sure it looks good. Thanks Javier!

Ok, let's take this for a test drive. First, we need a big title: $style->title('Welcome to SymfonyStyle!') and then a sub-header with $style->section('Wow, look at this text section');. To print out normal text, use $style->text(). I'll quote all demo pages on the Internet by saying "Lorem ipsum Dolor"... a bunch of times:

29 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 19
20          $style = new SymfonyStyle($input, $output);
21          $style->title('Welcome to SymfonyStyle');
22          $style->section('Wow, look at this text section');
23          $style->text('Lorem ipsum dolor, Lorem ipsum dolor, Lorem ipsum dolor, Lorem ipsum dolor, Lorem ipsum dolor, Lorem ipsum d
... lines 24 - 29
```

Because we're worried about all these "Lorem ipsums", use $style->note('') to remind people to "write some real text eventually":

29 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 23
24          $style->note('Make sure you write some *real* text eventually');
... lines 25 - 29
```

Let's try it! Run:

```
$ ./bin/console styles:play
```

OooOOooooOOO. Without doing anything, Javier gives us the console equivalent of h1 and h2 tags, with colors and separation. Below, it's subtle, but all the text lines are indented with one space to make them stand out. The note starts with an exclamation point and uses a hip color.

Get the idea?

Back on the command, add one more text line: $style->comment('Lorem ipsum is just latin garbage');. Follow that with $style->comment('So don\'t overuse it'). Make sure the method name is correct:

29 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 24
25          $style->comment('Lorem ipsum is just Latin garbage');
26          $style->comment('So don\'t overuse it');
... lines 27 - 29
```

Run it again!

```
$ ./bin/console styles:play
```

More indented text, this time with a little bit different styling.

Success and Error Messages

Time to take the fancy up a notch! Add another section for some BIG messages. SymfonyStyle has built-in methods to emphasize that "things are great!", "things are terrible! or "OMG things are *really* terrible". Start with $style->success('I <3 Lorem ipsum');:

35 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 27
        $style->section('How about some BIG messages?');
        $style->success('I <3 lorem ipsum');
... lines 30 - 35
```

Try it!

```
$ ./bin/console styles:play
```

A big ol' nice green message that just screams celebration.

Ok, maybe we don't love "Lorem ipsum". Send a warning with $style->warning('You should *maybe* not use Lorem ipsum');:

35 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 29
        $style->warning('You should *maybe* not use Lorem ipsum');
... lines 31 - 35
```

Try that!

```
$ ./bin/console styles:play
```

Now we have a menacing red message: you've been warned...

Next, try $style->error('You should stop using Lorem ipsum');. And throw in one last word of caution, $style->caution('STOP USING IT SRSLY!');:

35 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 30
        $style->error('You should stop using lorem ipsum');
        $style->caution('STOP USING IT SRSLY!');
... lines 33 - 35
```

When we run this, we've got four blocks: all styled for their meaning with nice spacing, coloration and margin. Thanks Javier!

Tables and Lists

The SymfonyStyle has helpers for a few other things, like progress bars and tables. Create a new section: $style->section('Some tables and lists?');:

51 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 33
        $style->section('Some tables and lists?');
... lines 35 - 51
```

Creating tables isn't new, but $style has a shortcut where Javier styles them for us. Thanks Javier!

Use $style->table(). The first argument holds the headers: ['User', 'Birthday'], and the second argument holds the rows. Plug in an important birthday to remember: ['weaverryan','June 5th'] and an even *more* important one to remember, ['leannapelham','All of February']. That's right, the celebration for Leanna never ends:

51 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 34
        $style->table(
            ['User', 'Birthday'],
            [
                ['weaverryan', 'June 5th'],
                ['leannapelham', 'All February']
            ]
        );
... lines 42 - 51
```

So let's see how this renders!

```
$ ./bin/console styles:play
```

Wow, look at that table: nice spacing, nice styled headers. Of course you could do this all yourself, but why?

Ok, time for just *one* more: a list of my favorite things: $style->text('Ryan\'s my favorite things') with a nicely-styled list. Use $style->listing([]). Pass this an array, which you can think of as an unordered list. Let's see, I like ['Running', 'Pizza', 'Watching Leanna tease Jordi Boggiano',]:

51 lines | src/AppBundle/Command/StylesPlayCommand.php

```
... lines 1 - 42
        $style->text('Ryan\'s favorite things:');
        $style->listing([
            'Running',
            'Pizza',
            'Watching Leanna tease Jordi Boggiano'
        ]);
... lines 49 - 51
```

Ok last run in the terminal!

```
$ ./bin/console styles:play
```

There you have it, the SymfonyStyle. Thanks Javier! Seriously, Javier is a cool dude.

Chapter 8: The new Voter Class

Create a new controller - viewUserAction - that will live at /users/{username} with a route named user_view:

73 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 10
class DefaultController extends Controller
{
... lines 13 - 60
    /**
     * @Route("/users/{username}", name="user_view")
     */
    public function viewUserAction(User $user)
    {
... lines 66 - 70
    }
}
```

Here's the challenge: we need to restrict who is allowed to view this page based on some complex business logic. Maybe I can view only *my* page... unless I'm an admin... who can view anyone's page. This is a classic situation where security isn't global, it's dependent on the object being accessed. I can see *my* user page but not *your* user page.

This is the perfect case for voters. I've been talking about these for years... and they are *still* underused. Repeat after me, "I do not need ACL, I need voters". And good news, voters are even easier to use in Symfony 3... I mean 2.8.

Instead of passing $username to the action, type-hint the User object:

73 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 10
class DefaultController extends Controller
{
... lines 13 - 60
    /**
     * @Route("/users/{username}", name="user_view")
     */
    public function viewUserAction(User $user)
    {
... lines 66 - 70
    }
}
```

Thanks to the FrameworkExtraBundle, Symfony will query for the User automatically based on the username property.

Using a Voter

Next, add if (!$this->isGranted('USER_VIEW', $user)){}. If this is not granted,
throw $this->createAccessDeniedException('No!'). If access *is* granted just dump('Access granted', $user);die;:

73 lines | src/AppBundle/Controller/DefaultController.php

```
... lines 1 - 63
    public function viewUserAction(User $user)
    {
        if (!$this->isGranted('USER_VIEW', $user)) {
            throw $this->createAccessDeniedException('NO!');
        }

        dump('Access granted!', $user);die;
    }
... lines 72 - 73
```

The mysterious thing is USER_VIEW. We *usually* pass things like ROLE_USER or ROLE_ADMIN to isGranted(). But you can invent whatever string you want: I made up USER_VIEW.

The Voter System

Whenever you call isGranted(), Symfony asks a set of "voters" whether or not the current user should be granted access. One of the default voters handles anything that starts with ROLE_. And guess what! You can *also* pass an object as a second argument to isGranted(). That's also passed to the voters.

Here's the plan: create a new voter that decides access whenever we pass USER_VIEW to isGranted().

Create the Voter

In the Security directory, create a new class called UserVoter and make this extend Voter:

19 lines | src/AppBundle/Security/UserVoter.php

```
... lines 1 - 2
namespace AppBundle\Security;
... lines 4 - 5
use Symfony\Component\Security\Core\Authorization\Voter\Voter;

class UserVoter extends Voter
{
... lines 10 - 18
}
```

This is a new class in Symfony 2.8 that's easier than the old AbstractVoter.

Use command+n to open the generate menu and select "Implement methods". The two methods you need are supports and voteOnAttribute:

19 lines | src/AppBundle/Security/UserVoter.php

```
... lines 1 - 4
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
... lines 6 - 7
class UserVoter extends Voter
{
    protected function supports($attribute, $object)
    {

    }

    protected function voteOnAttribute($attribute, $object, TokenInterface $token)
    {
        // TODO: Implement voteOnAttribute() method.
    }
}
```

This is a little different than before.

Now stop! And go register this as a service. Call it user_voter and add its class: UserVoter. There aren't any arguments, but you *do* need a tag called security.voter:

16 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 11
    user_voter:
        class: AppBundle\Security\UserVoter
        tags:
            - { name: security.voter }
```

As soon as we give it this tag, the supports method will be called *every* time we call isGranted(), asking our voter "Yo! Do you support this attribute, like ROLE_USER or USER_VIEW?".

I'm already logged in - so I'll head to /users/weaverryan. Access denied! Right now, *none* of the voters are voting on this: they are *all* saying that they don't support the USER_VIEW attribute. If nobody votes, access is denied.

Adding Voter Logic

So let's code: In supports(), if (attribute != 'USER_VIEW'), then return false:

35 lines | src/AppBundle/Security/UserVoter.php

```
... lines 1 - 8
class UserVoter extends Voter
{
... lines 11 - 17
    protected function supports($attribute, $object)
    {
        if ($attribute != 'USER_VIEW') {
            return false;
        }
... lines 23 - 28
    }
... lines 30 - 34
}
```

This says: "I don't know, go bother some other voter!".

Add another if statement. Wait! Change the argument to $object - this *is* the object - if any - that's passed to isGranted(). Some now-fixed bad PHP-Doc in Symfony caused that issue.

Anyways, if (!$object instanceof User), then also return false: we only vote on User objects. Finally, at the bottom, return true:

35 lines | src/AppBundle/Security/UserVoter.php

```
... lines 1 - 17
    protected function supports($attribute, $object)
    {
        if ($attribute != 'USER_VIEW') {
            return false;
        }

        if (!$object instanceof User) {
            return false;
        }

        return true;
    }
... lines 30 - 35
```

You can of course make your voter support multiple attributes like USER_EDIT or even multiple objects. I usually have one voter per object.

If you return true from supports(), then voteOnAttribute() is called:

35 lines | src/AppBundle/Security/UserVoter.php

```
... lines 1 - 8
class UserVoter extends Voter
{
... lines 11 - 30
    protected function voteOnAttribute($attribute, $object, TokenInterface $token)
    {
... line 33
    }
}
```

This is where you shine: do whatever crazy business logic you need to and ultimately return true for access or false to deny access. The $attribute and $object are the same as before and $token gives you access to the currently-logged-in user.

Instead of adding real logic, let's let EvilSecurityRobot decide our fate. Add a __construct() method with EvilSecurityRobot as the only argument. Create a property and set it:

35 lines | src/AppBundle/Security/UserVoter.php

```
... lines 1 - 8
class UserVoter extends Voter
{
    private $robot;

    public function __construct(EvilSecurityRobot $robot)
    {
        $this->robot = $robot;
    }
... lines 17 - 34
}
```

In voteOnAttribute(), return $this->robot->doesRobotAllowAccess();:

35 lines | src/AppBundle/Security/UserVoter.php

```
... lines 1 - 30
    protected function voteOnAttribute($attribute, $object, TokenInterface $token)
    {
        return $this->robot->doesRobotAllowAccess();
    }
```

Finally, update the service in services.yml for the new argument. But take the lazy way out: set autowire: true:

17 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 11
    user_voter:
        class: AppBundle\Security\UserVoter
        autowire: true
        tags:
            - { name: security.voter }
```

Head to the brower and try it!

Hey, access granted! Refresh again, access not granted! Again! Not granted, again, granted! The EvilSecurityRobot is hard at work causing us problems with its evil access rules.

Ok, tl;dr: use voters, they're easier than ever and you can do whatever crazy business logic you need.