Starting in Symfony2: Course 2 (2.4+)

With <3 from SymfonyCasts

Chapter 1: Introduction

INTRODUCTION¶

Well hi again! And congrats on getting through everything in episode 1. Seriously, that was a lot of work - so find a stranger and give them a high-five. Then come back and let's get to work.

Over the next hour, we're going to take things to the next level, aiming at some of the most difficult areas of Symfony, like security, forms, and some serious Doctrine topics. Some of this stuff will look pretty tough at first, but that's just because we're taking your Symfony foo up to the next level. When you finish, you'll be another huge step towards mastering Symfony.

Ok, drink some coffee, stretch a little, and lets get to it!

Chapter 2: Security Fundamentals

SECURITY FUNDAMENTALS¶

Symfony comes with a security component that's really powerful. Honestly, it's also really complex. It can connect with other authentication systems - like Facebook or LDAP - or load user information from anywhere, like a database or even across an API.

The bummer is that hooking all this up can be tough. But since you'll know how each piece works, you'll be able to do amazing things. There's also some jedi magic I'll show you later that makes custom authentication systems much easier.

Authentication, Authorization and the Death Star¶

Security is two parts: authentication and authorization. **Authentication, checks the user's credentials**. Its job is *not* to restrict access, it just wants to know *who* you are.

Ok, so think of a building, or maybe even the Death Star. After the tractor beam forces you to land, you walk out and pass through a security checkpoint. Both Stormtroopers *and* rebels check-in here, prove who they are and receive an access card, or a *token* in Symfony-speak.

Proving who you are and getting a token: that's authentication.

The token can be used to unlock doors in this fully armed and operational battle station. Everyone inside has a token, but some grant more access than others. The second part of security, authorization, is like the lock that's on every door. It actually *denies* a user access to something. Authorization doesn't care if you're Obi-Wan or a Stormtooper, it only checks to see if the token you received has enough access to enter a specific room.

Security configuration: security.yml¶

Let's talk authentication first, which can be more complex than authorization. The security configuration lives entirely in the `app/config/security.yml` file, which is imported from the main `config.yml` file:

```
# app/config.config.yml
imports:
    # ...
    - { resource: security.yml }
```

Security config lives in its own file because, well, it's kind of big and ugly. But there's no technical reason: you could move all of this into `config.yml` and it would work just the same.

Firewalls Configuration (security.yml)¶

Note

If your `security.yml` file is mostly empty, don't worry! You installed Symfony in a slightly different way. Just download the code for this tutorial and replace your `security.yml` file with the one from the download.

Find the `firewalls` key: it's the most important part in this file. A firewall represents the authentication layer, or security check-point for your app. Delete the `login` and `dev` firewall sections so that we have just one firewall:

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

firewalls:
    secured_area:
        pattern:    ^/demo/secured/
        form_login:
            check_path: _security_check
            login_path: _demo_login
        logout:
            path:   _demo_logout
            target: _demo
        #anonymous: ~
        #http_basic:
        #    realm: "Secured Demo Area"
```

Just like in a giant floating death machine, it make sense for everyone to pass through the same security system that looks up people in the same corrupt, imperial database. In fact, change the `pattern` key to be `^/`:

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

firewalls:
    secured_area:
        pattern:    ^/
        # ...
```

Now, every request that goes to our app will use this one firewall for authentication. Let's also change the `login_path` key to be `/my-login-url`:

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

firewalls:
    secured_area:
        pattern:    ^/
        form_login:
            check_path: _security_check
            login_path: /my-login-url
        # ...
```

Don't worry about what this or any of the other keys mean yet: they're just there to confuse you. I'll explain it all in a second.

Anonymous Access (security.yml)¶

Now, uncomment the `anonymous` key:

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

security:
    # ...
    firewalls:
        secured_area:
            pattern:    ^/
            # ...
            anonymous: ~
```

This lets anonymous users into the site, similar to letting a tourist enter the Death Star. We may want to require login for certain pages, or even maybe nearly every page. But we're not going to do that here. Remember, the firewall is all about

finding out *who* you are, not denying access.

Head back to the browser, but don't refresh! First, notice the little red icon on your web debug toolbar. When you hover over it, it says "You are not authenticated".

Now refresh. Yay! It's green and says "anon". Clicking it shows us that we're now "authenticated". Yes, it's a bit odd, but anonymous users are actually authenticated, since they passed through our firewall.

But don't panic, it's easy in code to check if the user has *actually* logged in or not. I'll show you later. Of course, we haven't actually done the work to make it possible to login yet, but we'll get to those silly details in a second.

Chapter 3: Authorization with Access Control

AUTHORIZATION WITH ACCESS CONTROL¶

Before we keep going with authentication and make it possible to login, let's try out our first piece of authorization and start denying access!

Head back to `security.yml`. The easiest way to deny access is via the `access_control` section. Let's use its regular expression coolness to protect any URLs that start with "/new" or "/create".

Roles are given to a user when they login and if you're not logged in, you don't have any. Here, we're saying that you at least need `ROLE_USER` to access these URLs:

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/new, roles: ROLE_USER }
        - { path: ^/create, roles: ROLE_USER }
```

Try it out! When we try to add an event, we're redirected to `/my-login-url`. Hey! I know that URL! That's what we put for the `login_path` config key.

So here's the magic that just happened behind the scenes:

1. We tried to go to `/new`. Since our anonymous user doesn't have any roles, the access controls kicked us out;
2. The firewall saves the day. Instead of giving us an access denied screen, it decides to give us a chance to login. The `form_login` key in tells the firewall that we want to use a good old fashioned login form, and that the login form should live at `/my-login-url`.

It's *our* job to actually create the login page. And since we haven't yet, we see the big ugly 404 error.

More access_control options¶

The `access_control` has a few more tricks to it. Head over to the Security chapter of the book and find the section on `access_control`. I want you to read this, but the most important thing to know is that only *one* `access_control` entry is matched on a request. Symfony goes down the list, finds the first match, and uses *only* it to check authorization. I'll show you an example during the *last chapter*.

There's also other goodies, like different access controls based on the user's IP address or depending on which hostname is being accessed. You can even make it so that a user is redirected to `https`.

Chapter 4: Creating a Login Form (Part 1)

CREATING A LOGIN FORM (PART 1)¶

So where's the actual login form? Well, that's our job - the security layer just helps us by redirecting the user here.

Oh, and there's a really popular open source bundle called FosUserBundle that gives you a lot of what we're about to build. The good news is that after building a login system in this tutorial, you'll better understand how it works. So build it once here, then take a serious look at `FosUserBundle`.

Creating a Bundle by Hand¶

Let's create a brand new shiny bundle called `UserBundle` for all of our user and security stuff. We *could* use the `app/console generate:bundle` task to create this, but let's do it by hand. Seriously, it's easy.

Just create a `UserBundle` directory and an empty `UserBundle` class inside of it. A bundle is nothing more than a directory with a bundle class:

```
// src/Yoda/UserBundle/UserBundle.php
namespace Yoda\UserBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class UserBundle extends Bundle
{
}
```

Now, just activate it in the AppKernel class and, voila! Our brand new shiny bundle is ready:

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

public function registerBundles()
{
    $bundles = array(
        // ...
        new Yoda\UserBundle\UserBundle(),
    );

    // ...
}
```

Login Form Controller¶

To make the login page, add a `Controller` directory and put a new `SecurityController` class inside of it. Give the class a `loginAction` method. This will render our login form:

```
// src/Yoda/UserBundle/Controller/SecurityController.php
namespace Yoda\UserBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class SecurityController extends Controller
{
    public function loginAction()
    {
    }
}
```

Using Annotation Routing¶

Before we fill in the guts of `loginAction`, we need a route! After watching episode 1, you probably expect me to create a `routing.yml` file in `UserBundle` and add a route there.

Ha! I'm not so predictable! Instead, we're going to get crazy and build our routes right inside the controller class using annotations. The docs for this feature live at symfony.com under a bundle called SensioFrameworkExtraBundle. This bundle came pre-installed in our project. How thoughtful!

First, add the `Route` annotation namespace:

```
// src/Yoda/UserBundle/Controller/SecurityController.php
// ...

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

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

Now, we can add the route right above the method:

```
// src/Yoda/UserBundle/Controller/SecurityController.php
// ...

/**
 * @Route("/login", name="login_form")
 */
public function loginAction()
{
    // ... todo still..
}
```

Finally, tell Symfony to look for routes in our controller by adding an import to the main `routing.yml` file:

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

user_routes:
    resource: "@UserBundle/Controller"
    type: annotation
```

Remember that Symfony never automatically finds routing files: we always import them manually from here.

Cool - change the URL in your browser to `/login`. This big ugly error about our controller not returning a response is great news! No, seriously, it means that the route is working. Now let's fill in the controller!

The loginAction Logic¶

Most of the login page code is pretty boilerplate. So let's use the age-old art of copy-and-paste from the docs.

Head to the security chapter and find the login form section. Copy the `loginAction` and paste it into our controller. Don't forget to add the `use` statements for the `SecurityContextInterface` and `Request` classes:

```
// src/Yoda/UserBundle/Controller/SecurityController.php
namespace Yoda\UserBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\Security\Core\SecurityContextInterface;
use Symfony\Component\HttpFoundation\Request;
// ...

class SecurityController extends Controller
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(Request $request)
    {
        $session = $request->getSession();

        // get the login error if there is one
        if ($request->attributes->has(SecurityContextInterface::AUTHENTICATION_ERROR)) {
            $error = $request->attributes->get(
                SecurityContextInterface::AUTHENTICATION_ERROR
            );
        } else {
            $error = $session->get(SecurityContextInterface::AUTHENTICATION_ERROR);
            $session->remove(SecurityContextInterface::AUTHENTICATION_ERROR);
        }

        return $this->render(
            'AcmeSecurityBundle:Security:login.html.twig',
            array(
                // last username entered by the user
                'last_username' => $session->get(SecurityContextInterface::LAST_USERNAME),
                'error'         => $error,
            )
        );
    }
```

The method *just* renders a login template: it doesn't handle the submit or check to see if the username and password are correct. Another layer handles that. It *does* pass the login error message to the template if there is one, but that's it.

The Template Annotation Shortcut¶

The pasted code is rendering a template using our favorite `render` method that lives in Symfony's base controller.

Hmm, let's *not* do this. Instead, let's use another shortcut: the @Template annotation, which is also from SensioFrameworkExtraBundle.

Anytime we use an annotation in a class for the first time, we'll need to add a `use` statement for it. Copy this from the docs. Now, put `@Template` above the method and just return the array of variables you want to pass to Twig:

```php
// src/Yoda/UserBundle/Controller/SecurityController.php
// ...

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

class SecurityController extends Controller
{
    /**
     * @Route("/login", name="login_form")
     * @Template()
     */
    public function loginAction()
    {
        // ...

        return array(
            // last username entered by the user
            'last_username' => $session->get(SecurityContextInterface::LAST_USERNAME),
            'error'         => $error,
        );
    }
}
```

With `@Template`, Symfony renders a template automatically, and passes the variables we're returning into it. It's cool, saves us some typing and supports the rebel forces.

Chapter 5: Creating a Login Form (Part 2)

CREATING A LOGIN FORM (PART 2)¶

Ok, we're almost done, seriously!

Creating the Template¶

Copy the template code from the docs and create the `login.html.twig` file:

```
{# src/Yoda/UserBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    {#
        If you want to control the URL the user
        is redirected to on success (more details below)
        <input type="hidden" name="_target_path" value="/account" />
    #}

    <button type="submit">login</button>
</form>
```

This prints the login error message if there is one and has a form with `_username` and `_password` fields. When we submit, Symfony is going to be looking for these fields, so their names are important.

Tip

You can of course change these form field names to something else. Google for the `username_parameter` and `password_parameter` options.

Let's make this extend our `base.html.twig` template. I'll also add in a little bit of extra markup:

```
{# src/Yoda/UserBundle/Resources/views/Security/login.html.twig #}
{% extends '::base.html.twig' %}

{% block body %}
<section class="login">
    <article>

        {% if error %}
            <div>{{ error.message }}</div>
        {% endif %}

        <form action="{{ path('login_check') }}" method="post">
            <label for="username">Username:</label>
            <input type="text" id="username" name="_username" value="{{ last_username }}" />

            <label for="password">Password:</label>
            <input type="password" id="password" name="_password" />

            <button type="submit">login</button>
        </form>

    </article>
</section>
{% endblock %}
```

Handling Login: login_check¶

Route check! Controller check! Template check! Let's try it! Oh boy, an error:

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

Ah, the copied template code has a form that submits to a route called `login_check`. Let's create another action method and use `@Route` to create that route:

```
// ...
// src/Yoda/UserBundle/Controller/SecurityController.php

/**
 * @Route("/login_check", name="login_check")
 */
public function loginCheckAction()
{
}
```

Call me crazy, but I'm going to leave this action method completely blank. Normally, it means that if you went to `/login_check` it would execute this controller and cause an error since we're not returning anything.

Configuring login_path and check_path¶

But this controller will never be executed. Before I show you, open up `security.yml` and look at the `form_login` configuration:

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

firewalls:
    secured_area:
        pattern:    ^/
        form_login:
            check_path: _security_check
            login_path: /my-login-url
        # ...
```

`login_path` is the URL *or* route name the user should be sent to when they hit a secured page. Change this to be `login_form`: the name of our `loginAction` route. `check_path` is the URL or route name that the login form will be submitted to. Change this to be `login_check`.

In your browser, try going to `/new`. Yes! *Now* we're redirected to `/login`, thanks to the `login_path` config key. The page looks just terrible, but it's working.

Using and Understanding the Login Process¶

Now, let me show you one of the strangest parts of Symfony's security system. When we login using `user` and `userpass` ... it works! We can see our username in the web debug toolbar and even a role assigned to us. What the heck just happened?

When we submit, Symfony's security system intercepts the request and processes the login information. This works as long as we POST `_username` and `_password` to the URL `/login_check`. This URL is special because its route is configured as the `check_path` in `security.yml`. The `loginCheckAction` method is *never* executed, because Symfony intercepts POST requests to that URL.

If the login is successful, the user is redirected to the page they last visited or the homepage. If login fails, the user is sent back to `/login` and an error is shown.

And where did the `user` and `userpass` stuff come from? Actually, right now the users are just being loaded directly from `security.yml`:

```
# app/config/security.yml
# ...
providers:
    in_memory:
        memory:
            # this was here when we started: 2 hardcoded users
            users:
                user:  { password: userpass, roles: [ 'ROLE_USER' ] }
                admin: { password: adminpass, roles: [ 'ROLE_ADMIN' ] }
```

In a minute, we'll load users from the database instead.

Chapter 6: Logging Out and Cleaning Up

LOGGING OUT AND CLEANING UP¶

What about logging out? Symfony has some magic for that too!

Look at the `logout` part of `security.yml`:

```
# app/config/security.yml
# ...
firewalls:
    secured_area:
        # ...
        logout:
            path:   _demo_logout
            target: _demo
```

`path` is the name of your logout route. Set it to `logout` - we'll create that route in a second. `target` is where you want to redirect the user after logging out. We already have a route called `event`, which is our event list page. Use that for `target`:

```
# app/config/security.yml
# ...
firewalls:
    secured_area:
        # ...
        logout:
            path:   logout # a route called logout
            target: event  # a route called event
```

To make the `logout` route, let's add another method inside `SecurityController` and use the `@Route` annotation:

```
// ...
// src/Yoda/UserBundle/Controller/SecurityController.php

/**
 * @Route("/logout", name="logout")
 */
public function logoutAction()
{
}
```

Just like with the `loginCheckAction`, the code here won't actually get hit. Instead, Symfony intercepts the request and processes the logout for us.

Try it out by going to `/logout` manually. Great! As you can see by the web debug toolbar, we're anonymous once again.

Cleaning up loginAction¶

If we fail login, we see a "Bad Credentials" message. When Symfony handles the login, it saves this error to the session under a special key, and we're just fetching it out in `loginAction`.

Actually, we have more code than we need here. Remove the if statement and just leave the second part:

```
// src/Yoda/UserBundle/Controller/SecurityController.php
// ...

public function loginAction(Request $request)
{
    $session = $request->getSession();

    // get the login error if there is one
    $error = $session->get(SecurityContextInterface::AUTHENTICATION_ERROR);
    $session->remove(SecurityContextInterface::AUTHENTICATION_ERROR);

    return array(
        // last username entered by the user
        'last_username' => $session->get(SecurityContextInterface::LAST_USERNAME),
        'error'         => $error,
    );
}
```

The first part isn't used unless you reconfigure how Symfony sends you to the login page.

Note

The configuration I'm talking about here is the `use_forward`, which causes Symfony to forward to the login page, instead of redirecting.

Adding CSS to a Single Page¶

I know I know, the login page is embarrassing looking. So I made a `login.css` file to fix things - find it in the `resources/episode2` directory of the code download.

Let's move it into a `Resources/public/css` directory in the `UserBundle`.

```
/* src/Yoda/UserBundle/Resources/public/css/login.css */
.login {
    width: 500px;
    margin: 100px auto;
}

/* for the rest of login.css, see the code download */
```

Just like in episode 1, run `app/console assets:install` and add the `--symlink` option, unless you're on Windows:

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

This creates a symbolic link from `web/bundles/user` to the `Resources/public` directory in UserBundle. Since `web/` is our application's document root, this makes our new CSS file accessible in a browser by going to `/bundles/user/css/login.css`.

So how can we add this CSS file to *only* this page? First, open up the base template. Here, we have a bunch of blocks, including one called `stylesheets`. All of our global CSS link tags live inside of it:

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

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

Let's override this block in `login.html.twig` and add the new link tag to `login.css`:

```
{# src/Yoda/UserBundle/Resources/views/Security/login.html.twig #}

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

Cool, but do you see the problem? This would entirely *replace* the block, but we want to *add* to it. The trick is the Twig parent() function. By including this, all the parent block's content is included first:

```
{# src/Yoda/UserBundle/Resources/views/Security/login.html.twig #}

{% block stylesheets %}
    {{ parent() }}

    <link rel="stylesheet" href="{{ asset('bundles/user/css/login.css') }}" />
{% endblock %}
```

Refresh now. Much less embarrassing looking. When you need to add CSS or JS to just one page, this is how you do it.

And by adding a little error class, it looks even better:

```
{# src/Yoda/UserBundle/Resources/views/Security/login.html.twig #}
{# ... #}

{% if error %}
    <div class="error">{{ error.message }}</div>
{% endif %}
```

And while we're making things look better, let's open up `base.html.twig` and add a link tag to the Bootstrap CSS file. Just use a CDN URL for simplicity:

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

{% block stylesheets %}
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css"/>

    ...
{% block stylesheets %}
```

Back in `login.html.twig`, I'll tweak the submit button so things look nicer:

```
{# src/Yoda/UserBundle/Resources/views/Security/login.html.twig #}
{# ... #}

<hr/>
<button type="submit" class="btn btn-primary pull-right">login</button>
```

Refresh! Ah, much better. I'm a programmer, but I don't want the site to look totally embarrassing!

Translating the Login Error Message¶

While we're here, let's change that "Bad Credentials" message, it's a little, "programmery". The message comes from deep inside Symfony. So to customize it, we'll use the translator.

First, use the Twig `trans` filter on the message:

```
{# src/Yoda/UserBundle/Resources/views/Security/login.html.twig #}
{# ... #}

{% if error %}
    <div class="error">{{ error.message|trans }}</div>
{% endif %}
```

Next, create a translation file in `app/Resources/translations/messages.en.yml`. This file is just a simple key-value pair of translations:

```
# app/Resources/translations/messages.en.yml
"Bad credentials": "Wrong password bro!"
```

Now, we just need to activate the translation engine in `app/config.yml`:

```
framework:
    # ...
    translator:     { fallback: %locale% }
```

Ok now, try it! Again, so much better!

Chapter 7: Twig Security and IS_AUTHENTICATED_FULLY

TWIG SECURITY AND IS_AUTHENTICATED_FULLY¶

Since logging out works, let's add a link to actually do it.

We already know logging out in Symfony is really easy. As long as the `logout` key is present under our firewall and we have a route to `/logout`, we can surf there and it'll just work. Symfony takes care of the details behind the scenes.

Security Inside Twig: is_granted¶

Open up the homepage template and add the logout link. This is just like generating any other URL: use the Twig `path` function and pass it the name of the route:

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

<a class="button" href="{{ path('event_new') }}">Create new event</a>

<a class="link" href="{{ path('logout') }}">Logout</a>

{# ... #}
```

It works of course, but we don't want to show it unless the user is *actually* logged in. To test for this, use the Twig is_granted function and pass it a special `IS_AUTHENTICATED_REMEMBERED` string:

```
{% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
    <a class="link" href="{{ path('logout') }}">Logout</a>
{% endif %}
```

And that works perfectly!

Trust Levels: IS_AUTHENTICATED_ANONYMOUSLY, IS_AUTHENTICATED_REMEMBERED, IS_AUTHENTICATED_FULLY¶

`is_granted` is how you check security in Twig, and we also could have passed normal roles here like `ROLE_USER` and `ROLE_ADMIN`, instead of this `IS_AUTHENTICATED_REMEMBERED` thingy. So in addition to checking to see if the user has a given role, Symfony has 3 other special security checks you can use.

- First, `IS_AUTHENTICATED_REMEMBERED` is given to all users who are logged in. They may have actually logged in during the session *or* may be logged in because they have a "remember me" cookie.
- Second, `IS_AUTHENTICATED_FULLY` is actually stronger. You only have this if you've *actually* logged in during this session. If you're logged in because of a remember me cookie, you won't have this;
- Finally, `IS_AUTHENTICATED_ANONYMOUSLY` is given to *all* users, even if you're not logged in. And since literally *everyone* has this, it seems worthless But it actually *does* have a use if you need to *white-list URLs that should be public*. I'll show you an example in the last chapter.

Since we're checking for `IS_AUTHENTICATED_REMEMBERED`, we're showing the logout link to anyone who is logged in, via a remember me cookie or because they recently entered their password. We want to let both types of users logout.

Let's get super fancy and add a login link for those anonymous souls:

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

{% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
    <a class="link" href="{{ path('logout') }}">Logout</a>
{% else %}
    <a class="link" href="{{ path('login_form') }}">Login</a>
{% endif %}
```

You'll probably want to use `IS_AUTHENTICATED_REMEMBERED` almost everywhere and save `IS_AUTHENTICATED_FULLY` for pages that need to be really secure, like checkout. If the user is *only* `IS_AUTHENTICATED_REMEMBERED` and hits one of those pages, they'll be redirected to login.

Chapter 8: Denying Access: AccessDeniedException

DENYING ACCESS: ACCESSDENIEDEXCEPTION¶

Let's login as `user` again and surf to `/new`. Since we have the `ROLE_USER` role, we're allowed access. In the `access_control` section of `security.yml`, change the role for this page to be `ROLE_ADMIN` and refresh:

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/new, roles: ROLE_ADMIN }
        # ...
```

This is the access denied page. It means that we *are* authenticated, but don't have access. Of course in Symfony's prod environment, we'll be able to customize how this looks. We'll cover how to customize error pages in the next episode.

The `access_control` section of `security.yml` is the easiest way to control access, but also the least flexible. Change the `access_control` entry back to use `ROLE_USER` and then comment both of them out. We're going to deny access from inside our controller class instead.

```
# app/config/security.yml
security:
    # ...
    access_control:
        # - { path: ^/new, roles: ROLE_USER }
        # - { path: ^/create, roles: ROLE_USER }
```

Denying Access From a Controller: AccessDeniedException¶

Find the `newAction` in `EventController`. To check if the current user has a role, we need to get the "security context". This is a scary sounding object, which has just one easy method on it: `isGranted`.

Use it to ask if the user has the `ROLE_ADMIN` role:

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

public function newAction()
{
    $securityContext = $this->container->get('security.context');
    if (!$securityContext->isGranted('ROLE_ADMIN')) {
        // panic?
    }

    // ...
}
```

If the user *doesn't* have `ROLE_ADMIN`, we need to throw a very special exception: called `AccessDeniedException`. Add a `use` statement for this class and then throw a new instance inside the `if` block. If you add a message, only the developers will see it:

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

use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function newAction()
{
    $securityContext = $this->container->get('security.context');
    if (!$securityContext->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException('Only an admin can do this!!!!')
    }

    // ...
}
```

In Symfony 2.5 and higher, there's event a shortcut `createAccessDeniedException` method:

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

if (!$securityContext->isGranted('ROLE_ADMIN')) {
    // in Symfony 2.5
    throw $this->createAccessDeniedException('message!');
}
```

When we refresh now, we see the access denied page. But if we were logged in as `admin`, who *does* have this role, we'd see the page just fine.

AccessDeniedException: The Special Class for Security¶

Normally, if you throw an exception, it'll turn into a 500 page. But the `AccessDeniedException` is special. First, if we're not already logged in, throwing this causes us to be redirected to the login page. But if we *are* logged in, we'll be shown the access denied 403 page. We don't have to worry about whether the user is logged in or not here, we can just throw this exception.

Phew! Security is hard, but wow, you seriously know almost everything you'll need to know. You'll only need to worry about the *really* hard stuff if you need to create a custom authentication system, like if you're authenticating users via an API key instead of a login form. If you're in this situation, make sure you read the Symfony Cookbook entry called How to Authenticate Users with API Keys. It uses a feature that's new to Symfony 2.4, so you may not see it mentioned in older blog posts.

Ok, let's unbreak our site. To keep things short, create a new private function in the controller called `enforceUserSecurity` and copy our security check into this:

```
private function enforceUserSecurity()
{
    $securityContext = $this->container->get('security.context');
    if (!$securityContext->isGranted('ROLE_USER')) {
        throw new AccessDeniedException('Need ROLE_USER!');
    }
}
```

Now, use this in `newAction`, `createAction`, `editAction`, `updateAction` and `deleteAction`:

```
public function newAction()
{
    $this->enforceUserSecurity();

    // ...
}

public function createAction(Request $request)
{
    $this->enforceUserSecurity();

    // ...
}
```

You can see how sometimes using `access_control` can be simpler, even if this method is more flexible. Choose whichever works the best for you in each situation.

Tip

You can also use annotations to add security to a controller! Check out SensioFrameworkExtraBundle.

Chapter 9: Entity Security

ENTITY SECURITY¶

Repeat after me, “We’re really great.” And our security system is almost as cool as we are. So let’s keep up the pace and load users from the database instead of the little list in `security.yml`.

What we’re about to do is similar to what the awesome open source FOSUserBundle gives you. We’re going to build this all ourselves so that we *really* understand how things work. Later, if you *do* use FOSUserBundle, you’ll be a lot more dangerous with it.

Generating the User Entity¶

Ok, forget about security! Seriously! Just think about the fact that we want to store some user information in the database. To do this, we’ll need a `User` entity class.

That sounds like a lot of work, so let’s just use the `doctrine:generate:entity` `app/console` command:

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

For entity shortcut name, use `UserBundle:User`. Remember, Doctrine uses this shortcut syntax for entities.

Give the class just 2 fields:

- `username` as a string
- `password` as a string

And of course, choose “yes” to generating the *repository class*. I’ll explain why these are so fabulous in a second.

Once the robots are done writing the code for us, we should have a new `User` class in the `Entity` directory of `UserBundle`. Let’s change the table name to be `yoda_user`:

```
// src/Yoda/UserBundle/Entity/User.php
namespace Yoda\UserBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table(name="yoda_user")
 * @ORM\Entity(repositoryClass="Yoda\UserBundle\Entity\UserRepository")
 */
class User
{
    // ... the generated properties and getter/setter functions
}
```

Implementing UserInterface¶

Right now, this is just a plain, regular Doctrine entity that has nothing to do with security. But, our goal is to load users from this table on login. The first step is to make your class implement a `UserInterface`:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

use Symfony\Component\Security\Core\User\UserInterface;

class User implements UserInterface
{
    // ...
}
```

This interface requires us to have 5 methods and hey! We already have 2 of them: `getUsername()` and `getPassword()`:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

public function getUsername()
{
    return $this->username;
}

public function getPassword()
{
    return $this->password;
}
```

Cool! So let's add the other 3.

First, `getRoles()` returns an array of roles that the user should get. For now, we'll hardcode a single role, `ROLE_USER`:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

public function getRoles()
{
    return array('ROLE_USER');
}
```

Second, add `eraseCredentials`. Keep this method blank for now. We will add some logic to this later:

```
public function eraseCredentials()
{
    // blank for now
}
```

Finally, add `getSalt()` and just make it return `null`:

```
public function getSalt()
{
    return null;
}
```

I'll talk more about this method in a second.

Now that the `User` class implements `UserInterface`, Symfony's authentication system will be able to use it. But before we hook that up, let's add the `yoda_user` table to the database by running the `doctrine:schema:update` command:

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

Loading Users from Doctrine: security.yml¶

And for the grand finale, let's tell the security system to use our entity class!

In `security.yml`, replace the encoder entry with *our* user class and set its value to `bcrypt`:

```
# app/config/security.yml
security:
    encoders:
        Yoda\UserBundle\Entity\User: bcrypt
    # ...
```

This tells Symfony that the `password` field on our `User` will be encoded using the bcrypt algorithm.

Installing password_compat¶

The one catch is that bcrypt isn't supported until PHP 5.5. So if you're using PHP 5.4 or lower, you'll need to install an extra library via Composer. No problem! Head to your terminal and use the composer `require` command and pass it `ircmaxell/password-compat`:

```
php composer.phar require ircmaxell/password-compat
```

When it asks, use the `~1.0.3` version. By the way, this `require` command is just a shortcut that updates our `composer.json` *for* us and then runs the Composer `update`:

```
"require": {
    "...": "..."
    "ircmaxell/password-compat": "~1.0.3"
},
```

Using the entity Provider¶

Now for the Jedi magic! In `security.yml`, remove the single `providers` entry and replace it with a new one:

```
# app/config/security.yml
security:
    # ...

    providers:
        our_database_users:
            entity: { class: UserBundle:User, property: username }
```

I'm just inventing the `our_database_users` part, that can be anything. But the `entity` key is a special built-in provider that knows how to load users via a Doctrine entity.

Yea, and that's really it! Ok, let's try it.

When you refresh, you *may* get an error:

```
There is no user provider for user "Symfony\Component\Security\Core\User\User".
```

Don't panic, this is just because we're still logged in as one of the hard-coded users... even though we just deleted them from `security.yml`. It's a one-time error - just refresh and it'll go away.

Creating and Saving Users¶

Chapter 10: Saving Users

SAVING USERS¶

Now that the error is gone, try logging in! Wait, but our user table is empty. So we can see the bad password message, but we can't *actually* log in yet.

But we're pros, so it's no problem. Let's copy the `LoadEvents` fixtures class (`LoadEvents.php`) into the `UserBundle` , rename it to `LoadUsers` , and update the namespaces:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
namespace Yoda\UserBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Yoda\UserBundle\Entity\User;

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

Saving users is *almost* easy: just create the object, give it a username and then persist and flush it.

The tricky part is that darn `password` field, which needs to be encoded with `bcrypt` :

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...
use Yoda\UserBundle\Entity\User;
// ...

public function load(ObjectManager $manager)
{
    $user = new User();
    $user->setUsername('darth');
    // todo - fill in this encoded password... ya know... somehow...
    $user->setPassword('');
    $manager->persist($user);

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

ContainerAwareInterface for Fixtures¶

But what's *cool* is that Symfony gives us an object that can do all that encoding for us. To get it, first make the fixture implement the `ContainerAwareInterface` :

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

use Symfony\Component\DependencyInjection\ContainerAwareInterface;

class LoadUsers implements FixtureInterface, ContainerAwareInterface
{
    // ...
}
```

This requires one new method - `setContainer`. In it, we'll store the `$container` variable onto a new `$container` property:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class LoadUsers implements FixtureInterface, ContainerAwareInterface
{
    private $container;

    // ...

    public function setContainer(ContainerInterface $container = null)
    {
        $this->container = $container;
    }
}
```

Because we implement this interface, Symfony calls this method and passes us the container object before calling `load`. Remember that the container is the array-like object that holds all the useful objects in the system. We can see a list of those object by running the `container:debug` console task:

```
php app/console container:debug
```

Encoding the Password¶

Let's create a helper function called `encodePassword` to you know encode the password! This step may look strange, but stay with me. First, we ask Symfony for a special "encoder" object that knows how to encrypt our passwords. Remember the `bcrypt` config we put in `security.yml`? Yep, this object will use that.

After we grab the encoder, we just call `encodePassword()`, grab a sandwich and let it do all the work:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

private function encodePassword(User $user, $plainPassword)
{
    $encoder = $this->container->get('security.encoder_factory')
        ->getEncoder($user)
    ;

    return $encoder->encodePassword($plainPassword, $user->getSalt());
}
```

Behind the scenes, it takes the plain-text password, generates a random salt, then encrypts the whole thing using bcrypt. Ok, so let's set this onto the `password` property:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

public function load(ObjectManager $manager)
{
    $user = new User();
    $user->setUsername('darth');
    $user->setPassword($this->encodePassword($user, 'darthpass'));
    $manager->persist($user);

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

Try it! Reload the fixtures from the command line:

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

Let's use the query console task to look at what the user looks like:

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

```
array (size=1)
  0 =>
    array (size=3)
      'id' => string '1' (length=1)
      'username' => string 'user' (length=4)
      'password' => string '$2y$13$BoVE3I5dmVkBjRp.l6uwyOl8Z8Ngokiaa.OUUuHoDbGDBdMRMUrmC' (length=60)
```

Nice! We can see the encoded password, which for `bcrypt`, also includes the randomly-generated `salt`. You *do* need to store the `salt` for each user, but with `bcrypt`, it happens automatically. Symfony requires us to have a `getSalt` function on our `User`, but it's totally not needed with `bcrypt`.

Back at the browser, we can login! Behind the scenes, here's basically what's happening:

1. A User entity is loaded from the database for the given username;
2. The plain-text password we entered is encoded with bcrypt;
3. The encoded version of the submitted password is compared with the saved password field. If they match, then you now have access to roam about this fully armed and operational battle station!

Chapter 11: Adding Dynamic Roles to each User

ADDING DYNAMIC ROLES TO EACH USER¶

Right now, all users get just one role: `ROLE_USER`, because it's what we're returning from the `getRoles()` function inside the `User` entity.

Add a `roles` field and make it a `json_array` type:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

/**
 * @ORM\Column(type="json_array")
 */
private $roles = array();
```

`json_array` allows us to store an array of strings in one field. In the database, these are stored as a JSON string. Doctrine takes care of converting back and forth between the array and JSON.

Now, just update the `getRoles()` method to use this property and add a `setRoles` method:

```
public function getRoles()
{
    return $this->roles;
}

public function setRoles(array $roles)
{
    $this->roles = $roles;

    // allows for chaining
    return $this;
}
```

Cool, except the way it's written now, a user could actually have *zero* roles. Don't let this happen: that user will become the undead and cause a zombie uprising. They can login, but they won't actually be authenticated. You've been warned.

Be a hero by adding some logic to `getRoles()`. Let's just guarantee that *every* user has `ROLE_USER`:

```
public function getRoles()
{
    $roles = $this->roles;
    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}
```

Update the SQL for the new field and then head back to the fixture file:

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

Let's copy the code here and make a second, `admin` user. Give this powerful Imperial user `ROLE_ADMIN`:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

public function load(ObjectManager $manager)
{
    // ...
    $manager->persist($user);

    $admin = new User();
    $admin->setUsername('wayne');
    $admin->setPassword($this->encodePassword($admin, 'waynepass'));
    $admin->setRoles(array('ROLE_ADMIN'));
    $manager->persist($admin);

    $manager->flush();
}
```

Let's reload the fixtures!

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

Now, when we login as admin, the web debug toolbar shows us that we have `ROLE_USER` *and* `ROLE_ADMIN`.

USING THE ADVANCEDUSERINTERFACE FOR INACTIVE USERS¶

Could we disable users, like if they were spamming our site? Well, we could just delete them and send a strongly-worded email, but yea, we can also disable them!

Add an `isActive` boolean field to User. If the field is false, it will prevent that user from authenticating. Don't forget to add the getter and setter methods either by using a tool in your IDE or by re-running the `doctrine:generate:entities` command:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

/**
 * @var bool
 *
 * @ORM\Column(type="boolean")
 */
private $isActive = true;

// ...
// write or generate your getIsActive and setIsActive methods...
```

After that, update our schema to add the new field:

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

So the `isActive` field *exists*, but it's not actually used during login. To make this work, change the `User` class to implement `AdvancedUserInterface` instead of `UserInterface`:

```php
// src/Yoda/UserBundle/Entity/User.php
// ...

use Symfony\Component\Security\Core\User\AdvancedUserInterface;

class User implements AdvancedUserInterface
{
    // ...
}
```

Tip

For the OO geeks, `AdvancedUserInterface extends UserInterface`.

The new interface is a stronger version of `UserInterface` that requires four additional methods. I'll use my IDE to generate these. If *any* of these methods return false, Symfony will block the user from logging in. To prove this, let's make them all return true except for `isAccountNonLocked`:

```php
// src/Yoda/UserBundle/Entity/User.php
// ...

public function isAccountNonExpired()
{
    return true;
}

public function isAccountNonLocked()
{
    return false;
}

public function isCredentialsNonExpired()
{
    return true;
}

public function isEnabled()
{
    return true;
}
```

Logging in now is less fun: we're blocked with a helpful message.

Each of these methods does the exact same thing: they block login. Each will give the user a different message, which you can *translate* if you want. Set each to return true, except for `isEnabled`. Let's have it return the value for our `isActive` property:

```php
// src/Yoda/UserBundle/Entity/User.php
// ...

public function isAccountNonLocked()
{
    return true;
}

public function isEnabled()
{
    return $this->getIsActive();
}
```

If `isActive` is `false`, this should prevent the user from logging in.

Head over to our user fixtures so we can try this. Set the admin user to inactive:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

public function load(ObjectManager $manager)
{
    // ...
    $admin->setIsActive(false);
    // ...
}
```

Next, reload your fixtures:

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

When we try to login, we're automatically blocked. Cool! Let's remove the `setIsActive` call we just added and reload the fixtures to put everything back where it started.

Chapter 12: Repository Security

REPOSITORY SECURITY¶

Now, let's have our users provide an email and let them login using it or their username.

Giving the User an Email¶

Let's start like we always do, by adding the property to the `User` class with some Doctrine annotations:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

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

Next, generate or write a getter and a setter for the new property. As a reminder, I'll use the `doctrine:generate:entities` command to do this:

```
php app/console doctrine:generate:entities UserBundle --no-backup
```

That little `--no-backup` prevents the command from creating a little backup version of the file. You're using version control, so we don't need to be overly cautions. You are using version control, right!!??

Next update the database schema to add the new field:

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

Finally, update the fixtures so that each user has an email:

```
// src/Yoda/UserBundle/DataFixtures/ORM/LoadUsers.php
// ...

public function load(ObjectManager $manager)
{
    // ...
    $user->setEmail('darth@deathstar.com');

    // ...
    $admin->setEmail('wayne@deathstar.com');

    // ...
}
```

Reload everything to refresh the database:

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

Doctrine Repositories¶

When a user logs in right now, the security system queries for it using the `username` field. That's because we told it to in our *security.yml configuration*. We could change it here to be `email` instead, but there's no way to say `email` *or* `username`. We

can make this more flexible. But first, we need to learn about Doctrine repositories.

Find and open `UserRepository`:

```
// src/Yoda/UserBundle/Entity/UserRepository.php
namespace Yoda\UserBundle\Entity;

use Doctrine\ORM\EntityRepository;

class UserRepository extends EntityRepository
{
}
```

This is a Doctrine repository, and it was generated for us. Every entity, has its own repository class and it knows that *this* is the repository class for `User` because of an annotation on that class:

```
/**
 * ...
 *
 * @ORM\Entity(repositoryClass="Yoda\UserBundle\Entity\UserRepository")
 */
class User implements AdvancedUserInterface, Serializable
```

Note

Actually, if you *don't* set the `repositoryClass` option, Doctrine just gives you a base repository class for that entity.

Repositories are where query logic should live. We could create methods like `findActiveUsers`, which would query the database for users that have a value of `1` for the `isActive` field.

Using Repositories¶

And actually, we've already been using repositories in our project. Open up `EventController` and check out the `indexAction` method:

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

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

    $entities = $em->getRepository('EventBundle:Event')->findAll();

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

The base EntityRepository and its Shortcuts¶

To query for Events, we call `getRepository` on the entity manager. The `getRepository` method actually returns an instance of our very own `EventRepository`. But when we open up that class, it's empty:

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

use Doctrine\ORM\EntityRepository;

class EventRepository extends EntityRepository
{
    // nothing here... boring!
}
```

So where does the `findAll` method live? The answer is Doctrine's base EntityRepository class, which we're extending. If we open it, you'll find some of the helpful methods that we talked about in the previous screencast, including `findAll()`. So *every* repository class comes with a few helpful methods to begin with.

To prove that `getRepository` returns *our* `EventRepository`, let's override the `findAll()` method and just `die` to see if our code is triggered:

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

class EventRepository extends EntityRepository
{
    public function findAll()
    {
        die('NOOOOOOOOO!!!!!!!!!');
    }
}
```

And when we go to the events page, our page gives us an epic cry.

The repositoryClass Option¶

Now, open up the Event entity. Above the class, you'll see an `@ORM\Entity` annotation:

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

/**
 * @ORM\Entity(repositoryClass="Yoda\EventBundle\Entity\EventRepository")
 */
class Event
```

Ah-hah! The `repositoryClass` is what's telling Doctrine to use `EventRepository`. Let's remove that part and see what happens:

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

/**
 * ...
 *
 * @ORM\Entity()
 */
class Event
```

When we refresh, there's no epic cry. In fact, everything works perfectly! We didn't tell Doctrine about our custom repository, so when we call `getRepository` in the controller, it just gives us an instance of the base `EntityRepository` class. That was nice! Our overridden `findAll` method is bypassed and the real one is used.

Let's undo our damage by re-adding the `repositoryClass` option and remove the dummy `findAll` method:

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

/**
 * @ORM\Entity(repositoryClass="Yoda\EventBundle\Entity\EventRepository")
 */
class Event
```

So every entity has its own repository with helpful methods like `findAll` for returning objects of that type. And when those shortcut methods won't work, we'll add our own methods. All of our query logic *should* live inside repositories - it'll make your life much more organized later.

Chapter 13: Doctrine's QueryBuilder

DOCTRINE'S QUERYBUILDER¶

What if we wanted to find a `User` by matching on the `email` *or* `username` columns? We would of course add a `findOneByUsernameOrEmail` method to `UserRepository`:

```
// src/Yoda/UserBundle/Entity/UserRepository.php
// ...

class UserRepository extends EntityRepository
{
    public function findOneByUsernameOrEmail()
    {
        // ... todo - get your query on
    }
}
```

To make queries, you can use an SQL-like syntax called DQL, for Doctrine query language. You can even use native SQL queries if you're doing something really complex.

But most of the time, I recommend using the awesome query builder object. To get one, call `createQueryBuilder` and pass it an "alias". Now, add the where clause with our `OR` logic:

```
// src/Yoda/UserBundle/Entity/UserRepository.php
// ...

public function findOneByUsernameOrEmail($username)
{
    return $this->createQueryBuilder('u')
        ->andWhere('u.username = :username OR u.email = :email')
        ->setParameter('username', $username)
        ->setParameter('email', $username)
        ->getQuery()
        ->getOneOrNullResult()
    ;
}
```

The query builder has every method you'd expect, like `leftJoin`, `orderBy` and `groupBy`. It's really handy.

The stuff inside `andWhere` looks similar to SQL except that we use "placeholders" for the two variables. Fill each of these in by calling `setParameter`. The reason this is separated into two steps is to avoid SQL injection attacks, which are really no fun.

To finish the query, call `getQuery` and then `getOneOrNullResult`, which, as the name sounds, will return the `User` object if it's found or null if it's not found.

Note

To learn more about the Query Builder, see doctrine-project.org: The QueryBuilder.

To try this out, let's temporarily reuse the EventController's `indexAction`. Get the `UserRepository` by calling `getRepository` on the entity manager. Remember, the argument you pass to `getRepository` is the entity's "shortcut name": the bundle name followed by the entity name:

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

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

    // temporarily abuse this controller to see if this all works
    $userRepo = $em->getRepository('UserBundle:User');

    // ...
}
```

Now that we have the UserRepository, let's try our new method and dump the result:

```
public function indexAction()
{
    // ...
    $userRepo = $em->getRepository('UserBundle:User');
    var_dump($userRepo->findOneByUsernameOrEmail('user'));die;

    // ...
}
```

When we refresh, we see the user. If we try the email instead, we get the same result:

```
var_dump($userRepo->findOneByUsernameOrEmail('user@user.com'));die;
```

Cool! Now let's get rid of these debug lines - I'm trying to get a working project going here people!

But this is a really common pattern we'll see more of: use the repository in a controller to fetch objects from the database. If you need a special query, just add a new method to your repository and use it.

Chapter 14: The UserProvider: Custom Logic to Load Security Users

THE USERPROVIDER: CUSTOM LOGIC TO LOAD SECURITY USERS¶

Hey there repository expert. So our *actual* goal was to let the user login using a username *or* email. If we could get the security system to use our shiny new `findOneByUsernameOrEmail` method to look up users at login, we'd be done. And back to our real job of crushing the rebel forces.

Open up `security.yml` and remove the `property` key from our entity provider:

```
# app/config/security.yml
security:
    # ...

    providers:
        our_database_users:
            entity: { class: UserBundle:User }
```

Try logging in now! Ah, a great error:

> The Doctrine repository "Yoda\UserBundle\Entity\UserRepository" must implement UserProviderInterface.

The UserProviderInterface¶

Without the property, Doctrine has no idea how to look up the User. Instead it tries to call a method on our `UserRepository`. But for that to work, our `UserRepository` class must implement `UserProviderInterface`.

So let's open up `UserRepository` and make this happen:

```
// src/Yoda/UserBundle/Entity/UserRepository.php
// ...

use Symfony\Component\Security\Core\User\UserProviderInterface;

class UserRepository extends EntityRepository implements UserProviderInterface
{
    // ...
}
```

As always, don't forget your `use` statement! This interface requires 3 methods: `refreshUser`, `supportsClass` and `loadUserByUsername`. I'll just paste these in:

```php
// src/Yoda/UserBundle/Entity/UserRepository.php
// ...

use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\Exception\UnsupportedUserException;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;

class UserRepository extends EntityRepository implements UserProviderInterface
{
    // ...

    public function loadUserByUsername($username)
    {
        // todo
    }

    public function refreshUser(UserInterface $user)
    {
        $class = get_class($user);
        if (!$this->supportsClass($class)) {
            throw new UnsupportedUserException(sprintf(
                'Instances of "%s" are not supported.',
                $class
            ));
        }

        if (!$refreshedUser = $this->find($user->getId())) {
            throw new UsernameNotFoundException(sprintf('User with id %s not found', json_encode($user->getId())));
        }

        return $refreshedUser;
    }

    public function supportsClass($class)
    {
        return $this->getEntityName() === $class
            || is_subclass_of($class, $this->getEntityName());
    }
}
```

Tip

You can get this code from the `resources` directory of the code download.

Filling in loadUserByUsername¶

The really important method is `loadUserByUsername` because Symfony calls it when you login to get the `User` object for the given username. So we can use any logic we want to find or not find a user, like never returning User's named "Jar Jar Binks":

```php
public function loadUserByUsername($username)
{
    if ($username == 'jarjarbinks') {
        // nope!
        return;
    }
}
```

We can just re-use the `findOneByUsernameOrEmail` method we created earlier. If no user is found, this method should throw a special `UsernameNotFoundException`:

```
// src/Yoda/UserBundle/Entity/UserRepository.php
// ...

class UserRepository extends EntityRepository implements UserProviderInterface
{
    // ...

    public function loadUserByUsername($username)
    {
        $user = $this->findOneByUsernameOrEmail($username);

        if (!$user) {
            throw new UsernameNotFoundException('No user found for username '.$username);
        }

        return $user;
    }

    // ... refreshUser and supportsClass from above...
}
```

Try logging in again using the email address. It works! Behind the scenes, Symfony calls the `loadUserByUsername` method and passes in the username we submitted. We return the right `User` object and then the authentication just keeps going like normal. We don't have to worry about checking the password because Symfony still does that for us.

Ok, enough about security and Doctrine! But give yourself a high-five because you just learned some of the most powerful, but difficult stuff when using Symfony and Doctrine. You now have an elegant form login system that loads users from the database and that gives you a lot of control over exactly how those users are loaded.

Now for a registration page!

Chapter 15: User Serialization

USER SERIALIZATION¶

There's a problem.

I have to bother you *quickly* with a little issue of serialization. When a user logs in, the `User` entity is stored in the session. For this to work, PHP serializes the User object to a string at the end of the request and stores it. At the beginning of the request, that string is unserialized and turned back into the User object.

Note

If you're feeling really curious, the class that serializes and deserializes the user information is called `ContextListener`.

This is great! And it's obviously working great - we're surfing around as Wayne the admin. But there's a "gotcha" in Doctrine. Sometimes, Doctrine will stick some extra information onto our entity, like the entity manager: that big important object we used to save things.

Normally, we don't care about this, but when the User object is serialized, having that big object hidden in our entity causes serialization to fail. The entity manager contains a database connection and other information that just *can't* be serialized.

Using the Serializable Interface¶

We need to help Doctrine out. Start by adding the `Serializable` interface to the User class. This core PHP interface has two methods: `serialize` and `unserialize`:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

use Serializable;

class User implements AdvancedUserInterface, Serializable
{
    // ...

    public function serialize()
    {
        // todo - do some mad serialization
    }

    public function unserialize($serialized)
    {
        // todo - and some equally angry de-serialization
    }
}
```

When the `User` object is serialized, it'll call the `serialize` method instead of trying to do it automatically. When the string is deserialized, the `unserialize` method is called. This may seem odd, but let's just return the `id`, `username` and `password` inside an array for `serialize`. For `unserialize`, just put those 3 values back on the object:

```php
// src/Yoda/UserBundle/Entity/User.php
// ..

public function serialize()
{
    return serialize(array(
        $this->id,
        $this->username,
        $this->password,
    ));
}

public function unserialize($serialized)
{
    list (
        $this->id,
        $this->username,
        $this->password,
    ) = unserialize($serialized);
}
```

If you think about it, this should kinda break everything. When Symfony gets the `User` object from the session and deserializes it, our User will have lost some of its data, like `roles` and `isActive`. That's not cool!

Clearly that's not the case: Symfony's security system is smart enough to take the `id` and query for a full fresh copy of the User object on each request.

We can see this right in the web debug toolbar: once a user is logged in, each request has a query that grabs the current user from the database. So, we're good!

Chapter 16: Registration Form

REGISTRATION FORM¶

Let's make our site a little bit more legit with a registration form. Go grab some coffee, cause we're about to rock our world with some big and powerful concepts like forms and validation.

Creating the Registration Page¶

Let's start by creating a new `RegisterController` class in UserBundle. Creating a controller by hand is easy: just add the right namespace and then extend Symfony's base controller class. The `registerAction` method will be our actual registration page:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
namespace Yoda\UserBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class RegisterController extends Controller
{
    public function registerAction()
    {
        // todo
    }
}
```

I'm kinda liking these annotation routes, so let's use those again. Remember, to do this, we need two things. First, add the Route use statement and then setup the route above the method:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class RegisterController extends Controller
{
    /**
     * @Route("/register", name="user_register")
     */
    public function registerAction()
    {
        // todo
    }
}
```

Second, we need to import the routes in `routing.yml`. And hey! We're already importing annotations from the entire `Controller/` directory, so we're ready to go!

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

user_routes:
    resource: "@UserBundle/Controller"
    type: annotation
```

To see if the route is being loaded, run the `router:debug` console task.

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

Yep, there it is!

Building the Form¶

Now let's build a form, which is an object that knows all of the fields we want and their types. It'll help us render the form and process its values. It's pretty fancy!

Start by calling the `createFormBuilder()` method. Now, use the `add` function to give the form `username`, `email` and `password` fields:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

public function registerAction()
{
    $form = $this->createFormBuilder()
        ->add('username', 'text')
        ->add('email', 'text')
        ->add('password', 'password')
        ->getForm()
    ;

    // todo next - render a template
}
```

The two arguments to `add` are the name of the field and the field "type". Symfony comes with built-in types for creating text fields, select fields, date fields and forcefields. I'll show you where to find a list in a minute.

When we're all done, we call `getForm()`.

Passing the Form into Twig¶

I want to render the form, so let's pass it to Twig. To save and impress my ewok friends, I'm going use the *@Template annotation trick* we saw earlier. Pass the form as the only variable:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

/**
 * @Route("/register", name="user_register")
 * @Template
 */
public function registerAction()
{
    $form = $this->createFormBuilder()
        // ...
        ->getForm()
    ;

    return array('form' => $form);
}
```

Tip

The above code has some bugs! Yuck! Keep reading below to fix them.

Fixing the Missing @Template Annotation¶

So let's create the Twig template. Make a "Register" directory in `Resources/views` since we're rendering from `RegisterController` and `@Template` uses that to figure out the template path. I'll paste in some HTML-goodness to get us started:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{% extends '::base.html.twig' %}

{% block stylesheets %}
    {{ parent() }}

    <link rel="stylesheet" href="{{ asset('bundles/user/css/login.css') }}" />
{% endblock %}

{% block body %}
<section class="login">
    <article>
        <h1>Register</h1>

    </article>
</section>
{% endblock %}
```

Tip

You can find this template code in the `resources/episode2` directory of the code download. Go get it!

So let's head to the browser to see how things look so far. When we go to `/register`, we see a nice looking page. Kidding! We see a huge, horrible threatening error!

> AnnotationException: [SemanticalError] The annotation "@Template" in method Yoda\UserBundle\Controller\RegisterController::registerAction() was never imported. Did you maybe forget to add a "use" statement for this annotation?

Tip

Sometimes errors are nested, and the most helpful parts are further below.

Look closely, the error contains the answer. Ah, I've used the `@Template` shortcut but forgot to put a `use` statement for it. After adding the namespace, I can refresh and see the page:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
```

Chapter 17: Form Rendering

FORM RENDERING¶

Create an HTML `form` tag that submits right back to the same route and controller we just created. The easiest way to render a form is all at once by using a special `form_widget` Twig function. Give it the form variable that we passed into the template, and add a submit button:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

{% block body %}
<section class="main-block">
  <article>
    <section>
      <h1>Register</h1>

      <form action="{{ path('user_register') }}" method="POST">
        {{ form_widget(form) }}

        <input type="submit" class="btn btn-primary pull-right" value="Register!" />
      </form>
    </section>
  </article>
</section>
{% endblock %}
```

Form versus FormView¶

Refresh the page. Woh man, another error!

> An exception has been thrown during the rendering of a template ("Catchable Fatal Error: Argument 1 passed to Symfony\Component\Form\FormRenderer::searchAndRenderBlock() must be an instance of Symfony\Component\Form\FormView, instance of Symfony\Component\Form\Form given, called in ...")

This one is more difficult to track down, but it *does* have one important detail: Specifically:

> Argument 1 passed to FormRenderer::searchAndRenderBlock() must be an instance of FormView, instance of Form given

That's a lot of words but it means that somewhere, we're calling a method and passing it the wrong type of object. It's expecting a `FormView`, but we're passing it a `Form`. Something is wrong with the form we created. Head back to `RegisterController` to fix this. Whenever you pass a form to a template, you *must* call `createView` on it. This transforms the object from a `Form` into a `FormView`:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

public function registerAction()
{
  // ...

  return array('form' => $form->createView());
}
```

This isn't important... just remember to do it. The error is a bit tough, but now you know it!

Refresh now and celebrate. We have a fully operational and rendred form. Actually, we haven't written any code to handle the form submit - that'll come in a minute.

Being event Lazier with form() and button¶

There's an even *easier* way to render the form, and while I don't love it, I want you to see it. Head to the Forms Chapter of the docs. Under the Rendering the Form section, it renders it with just one line:

```
{{ form(form) }}
```

This displays the fields, the HTML form tag and even a submit button, if you configure one. If you scroll up, you'll see that the form has a `save` field that's a `submit` type:

```
$form = $this->createFormBuilder($task)
    // ...
    ->add('save', 'submit')
    ->getForm();
```

I'm happy rendering the HTML form tags and the submit buttons myself, but you will see this rendering syntax, and I don't want you to be confused. It's all basically doing the same thing.

Rendering the Form one Field at a Time¶

In reality, rendering the form all at once probably won't be flexible enough in most cases. To render each field individually, use the `form_row` function on each of your fields:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

<form action="{{ path('user_register') }}" method="POST">
    {{ form_row(form.username) }}
    {{ form_row(form.email) }}
    {{ form_row(form.password) }}

    <input type="submit" class="btn btn-primary pull-right" value="Register!" />
</form>
```

Refresh the page and inspect the form. Each field row is surrounded by a `div` and contains the label and input:

```
<div>
    <label for="form_username" class="required">Username</label>
    <input type="text" id="form_username" name="form[username]" required="required" />
</div>
<!-- ... -->
```

Using form_widget, form_label and form_errors¶

In the next screencast, we'll learn how to customize how a field row is rendered. But even now, we can take more control by using the `form_label`, `form_widget` and `form_errors` functions individually. Let's try it on *just* the username field:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

<form action="{{ path('user_register') }}" method="POST">
    <div class="awesome-username-wrapper">
        {{ form_errors(form.username) }}
        {{ form_label(form.username) }}
        {{ form_widget(form.username) }}
    </div>

    {{ form_row(form.email) }}
    {{ form_row(form.password) }}

    <input type="submit" class="btn btn-primary pull-right" value="Register!" />
</form>
```

`form_row` just renders these 3 parts automatically, so this is basically the same as before. I usually try to use `form_row` whenever possible, so let's change the `username` back to use this.

Don't forget form_errors and form_rest!¶

Apart from the fields themselves, there are two other things that should be in every form. First, make sure you call `form_errors` on the entire form object:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

<form action="{{ path('user_register') }}" method="POST">
    {{ form_errors(form) }}

    {{ form_row(form.username) }}
    {{ form_row(form.email) }}
    {{ form_row(form.password) }}

    <input type="submit" class="btn btn-primary pull-right" value="Register!" />
</form>
```

Most errors appear next to the field they belong to. But in some cases, you might have a "global" error that doesn't apply to any one specific field. It's not common, but this takes care of rendering those.

Next, add `form_rest`. It renders any fields that you forgot:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

<form action="{{ path('user_register') }}" method="POST">
    {{ form_errors(form) }}

    {{ form_row(form.username) }}
    {{ form_row(form.email) }}
    {{ form_row(form.password) }}

    {{ form_rest(form) }}

    <input type="submit" class="btn btn-primary pull-right" value="Register!" />
</form>
```

In addition to that, `form_rest` is really handy because it renders any hidden fields automatically.

All forms have a hidden "token" field by default to protect against CSRF attacks. With `` `form_rest` ``, you never have to worry or think about hidden fields.

We talk more about these functions in future episodes, but under the reference section of Symfony's documentation, there's a page called Twig Template Form Function and Variable Reference that mentions all of these functions and how to use them.

Chapter 18: Using More Fields: email and repeated

USING MORE FIELDS: EMAIL AND REPEATED¶

We have just one password box, so lets turn this into 2 boxes by using the `repeated` field type. Let's also change the `email` field to be an `email` type:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

public function registerAction()
{
    $form = $this->createFormBuilder()
        ->add('username', 'text')
        ->add('email', 'email')
        ->add('password', 'repeated', array(
            'type' => 'password',
        ))
        ->getForm()
    ;

    // ..
}
```

The `repeated` field type is special because it actually renders *two* fields, in this case, password fields. If the two values don't match, a validation error will show up. If you refresh, you'll see the 2 fields. Oh no, attack of the clones!

The `email` field looks the same, but if you inspect it, you'll see that it's an input `email` field, an HTML5 field type that should be used:

```
<input type="email" ... />
```

Head over to Symfony's documentation and go into the Reference Section. There, you'll find a page called Form Field Type Reference. This is awesome: it shows you *all* of the built-in field types and the options you can pass to each. For example, if you click `repeated`, it shows you how to customize the error message that shows up if the fields don't match and some other stuff. Use this section to your advantage!

The Repeated Fields and "Compound" fields¶

Now look back at our 2 password fields. This highlights a very special aspect about the way forms work. Specifically, a single field may in fact be one or *many* fields:

```
<div>
    <!-- -->
    <input type="password" id="form_password_first" name="form[password][first]" required="required" />
</div>

<div>
    <!-- -->
    <input type="password" id="form_password_second" name="form[password][second]" required="required" />
</div>
```

When you use the repeated field type, it creates two sub-fields called "first" and "second". To see what I'm talking about, replace the `form_row` that renders the `password` field with two lines: one that renders the first box and one that renders the second:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

{{ form_row(form.username) }}
{{ form_row(form.email) }}
{{ form_row(form.password.first) }}
{{ form_row(form.password.second) }}

{# ... #}
```

Note

When a field is actually several fields, it's called a compound field.

When we refresh, we see exactly the same thing. I just wanted to highlight how `password` is really now *two* fields, and we can render them individually or both at once.

If this feels confusing, don't worry! This concept is a little bit more advanced.

Customizing Field Labels¶

Since "first" and "second" are, well, terrible labels, let's change them! One way to do this is by adding a second argument to `form_row` and passing a `label` key:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

{{ form_row(form.password.first, {
    label: 'Password'
}) }}

{{ form_row(form.password.second, {
    label: 'Repeat Password'
}) }}
```

Refresh! Much better!

Chapter 19: Handling Form Submissions

HANDLING FORM SUBMISSIONS¶

Ok, let's get this form to actually submit! Since we're submitting back to the same route and controller, we want to process things only if the request is a POST.

Getting the Request object¶

First, we'll need Symfony's Request object. To get this in a controller, add a `$request` argument that's type-hinted with Symfony's `Request` class:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

use Symfony\Component\HttpFoundation\Request;

class RegisterController extends Controller
{
    // ...
    public function registerAction(Request $request)
    {
        // ...
    }
}
```

Normally, if you have an argument here, Symfony tries to populate it from a routing wildcard with the same name as the variable. If it doesn't find one, it throws a giant error. The only exception to that rule is this: if you type-hint an argument with the Request class, Symfony will give you that object. This doesn't work for everything, only the Request class.

Using handleRequest¶

Use the form's `handleRequest` method to actually process the data. Next, add an `if` statement that checks to see if the form was submitted and if all of the data is valid:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

use Symfony\Component\HttpFoundation\Request;
// ...

public function registerAction(Request $request)
{
    $form = $this->createFormBuilder()
        // ...
        ->getForm()
    ;

    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {

        // do something in a moment
    }

    return array('form' => $form);
}
```

The `handleRequest` method grabs the POST'ed data from the request, processes it, and runs any validation. And actually, it

only does this for POST requests so on a GET request, `$form->isSubmitted()` returns false.

Tip

If you have a form that's submitted via a different HTTP method, set the method.

If the form *is* submitted because it's a POST request *and* passes validation, let's just print the submitted data for now. If the form is invalid, or if this is a GET request, it'll just skip this block and re-render the form with errors if there are any:

```
if ($form->isSubmitted() && $form->isValid()) {
    var_dump($form->getData());die;
}
```

Time to test it! We haven't added validation yet, but the password fields have built-in validation if the values don't match. When I submit, the form is re-rendered, meaning there was an error.

In fact, there's now a little red box on the web debug toolbar. If we click it, we can see details about the form: what was submitted and options for each field.

Head back and fill in the form correctly. Now we see our dumped data:

```
array(
    'username' => string 'foo' (length=3),
    'email' => string 'foo@foo.com' (length=11),
    'password' => string 'foo' (length=3),
)
```

Using the Submitted Data Array¶

Notice that the data is an array with a key and value for each field. Let's take this data and build a new `User` object from it. There *is* an easier way to do this, and I'll show you in a second:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

use Yoda\UserBundle\Entity\User;
// ...

$form->handleRequest($request);
if ($form->isSubmitted() && $form->isValid()) {
    $data = $form->getData();

    $user = new User();
    $user->setUsername($data['username']);
    $user->setEmail($data['email']);
}
```

Encoding the User's Password¶

We still need to encode and set the password. For now, let's copy in some code from our user fixtures to help with this. We'll make this much more awesome in the next screencast:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

private function encodePassword(User $user, $plainPassword)
{
    $encoder = $this->container->get('security.encoder_factory')
        ->getEncoder($user)
    ;

    return $encoder->encodePassword($plainPassword, $user->getSalt());
}
```

Use this function, then finally persist and flush the new User:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

if ($form->isValid()) {
    $data = $form->getData();

    $user = new User();
    $user->setUsername($data['username']);
    $user->setEmail($data['email']);
    $user->setPassword($this->encodePassword($user, $data['password']));

    $em = $this->getDoctrine()->getManager();
    $em->persist($user);
    $em->flush();

    // we'll redirect the user next...
}
```

Redirecting after Success¶

The last step of any successful form submit is to redirect - we'll redirect to the homepage. First, we need to generate a URL - just like we do with the `path()` function in Twig. In a controller, there's a `generateUrl` function that works exactly the same way:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

if ($form->isSubmitted() && $form->isValid()) {
    // ...

    $em->flush();

    $url = $this->generateUrl('event');
}
```

To redirect, use the `redirect` function and pass it the URL:

```
if ($form->isSubmitted() && $form->isValid()) {
    // ...
    $url = $this->generateUrl('event');

    return $this->redirect($url);
}
```

Tip

If you use Symfony `2.6` or newer, you have a `redirectToRoute` method that allows you to redirect based on the route name instead of having to generate the URL first. It's a shortcut!

Remember that a controller always returns a Response object. `redirect` is just a shortcut to create a Response that's all setup to redirect to this URL.

Ok, time to kick this proton torpedo! As expected, we end up on the homepage. We can even login as the new user!

You Don't Need isSubmitted¶

Head back to the controller and remove the `isSubmitted()` call in the `if` statement:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

$form->handleRequest($request);
if ($form->isValid()) {
    // ...
}
```

This actually doesn't change anything because `isValid()` automatically returns false if the form wasn't submitted - meaning, if the request isn't a POST. So either just do this, or keep the `isSubmitted` part in there if you want - I find it adds some clarity.

Chapter 20: Form: Default Data

FORM: DEFAULT DATA¶

Now, what if we wanted some default data to appear on the form? Well, we can just pass the data as the first argument to `createFormBuilder`:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

public function registerAction(Request $request)
{
    $defaultData = array(
        'username' => 'Leia',
    );

    $form = $this->createFormBuilder($defaultData)
        // ...
        ->getForm()
    ;

    // ...
}
```

Refresh and check that out.

HAVING THE FORM TO A USER OBJECT: THE DATA_CLASS OPTION¶

When we submit, `$form->getData()` gives us an associative array. That's cool, but what if it actually built the `User` object for us? Remove the default data we just added and pass a second argument to `createFormBuilder`. This is an array of options for the form and we'll pass it a `data_class` key that's set to our `User` class:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

public function registerAction(Request $request)
{
    $form = $this->createFormBuilder(null, array(
        'data_class' => 'Yoda\UserBundle\Entity\User',
    ))
        // ...
        ->getForm()
    ;

    // ...
}
```

Let's dump the form values again and try it:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

if ($form->isValid()) {
    $data = $form->getData();
    var_dump($data);die;

    // all the User saving code from before ...
}
```

Cool! Instead of an associative array, we get back a full `User` object populated with the form data. Behind the scenes, the form creates a new `User` object and then calls `setUsername`, `setEmail` and `setPassword` on it, passing each the value from the form.

Now, We can simplify things on our controller:

```
// inside registerAction()
if ($form->isValid()) {
    $user = $form->getData();

    $user->setPassword(
        $this->encodePassword($user, $user->getPassword())
    );

    $em = $this->getDoctrine()->getManager();
    // save the user and redirect just as before
}
```

DEFAULT DATA WITH AN OBJECT¶

So how can we set default data on the form now? Put back the array we had earlier:

```
$defaultData = array(
    'username' => 'Leia',
);

$form = $this->createFormBuilder($defaultData, array(
    'data_class' => 'Yoda\UserBundle\Entity\User',
))
    // ...
    ->getForm()
;
```

Refresh and look at the error message closely:

```
The form's view data is expected to be an instance of class
Yoda\UserBundle\Entity\User, but is a(n) array. You can avoid
this error by setting the "data_class" option to null or by adding
a view transformer that transforms a(n) array to an instance
of Yoda\UserBundle\Entity\User.
```

It's telling us that we gave the form an array but it was expecting a `User` object. The `data_class` option tells the form that both the output *and* the input of the form should be a `User`. So to set default data, just create a `User` object, give it some data and pass it in:

```
$user = new User();
$user->setUsername('Leia');

$form = $this->createFormBuilder($user, array(
    'data_class' => 'Yoda\UserBundle\Entity\User',
))
    // ...
    ->getForm()
;
```

Refresh now! It looks great!

Chapter 21: Cleaning up with a plainPassword Field

CLEANING UP WITH A PLAINPASSWORD FIELD¶

We're abusing our `password` field. It temporarily stores the plain text submitted password and then later stores the encoded version. This is a bad idea. What if we forget to encode a user's password? The plain-text password would be saved to the database instead of throwing an error. And storing plain text passwords is definitely against the Jedi Code!

Instead, create a new property on the `User` entity called `plainPassword`. Let's also add the getter and setter method for it:

```
private $plainPassword;

// ...

public function getPlainPassword()
{
    return $this->plainPassword;
}

public function setPlainPassword($plainPassword)
{
    $this->plainPassword = $plainPassword;

    return $this;
}
```

This property is just like the others, except that it's not actually persisted to the database. It exists just as a temporary place to store data.

Using eraseCredentials¶

Find the `eraseCredentials` method and clear out the `plainPassword` field:

```
public function eraseCredentials()
{
    $this->plainPassword = null;
}
```

This method isn't really important, but it's called during the authentication process and its purpose is to make sure your User doesn't have any sensitive data on it.

Using plainPassword¶

Now, update the form code - changing the field name from `password` to `plainPassword`:

```
// src/Yoda/UserBundle/Controller/RegisterController.php
// ...

public function registerAction(Request $request)
{
    // ...
    $form = $this->createFormBuilder(...)
        // ...
        ->add('plainPassword', 'repeated', array(
            'type' => 'password',
        ))
        ->getForm()
    ;

    // ...
}
```

Also don't forget to update the template:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

{{ form_row(form.plainPassword.first, {
    'label': 'Password'
}) }}

{{ form_row(form.plainPassword.second, {
    'label': 'Repeat Password'
}) }}
```

Now, when the form submits, the `plainPassword` is populated on the User. Use it to set the real, encoded `password` property:

```
// inside registerAction()
$user->setPassword(
    $this->encodePassword($user, $user->getPlainPassword())
);
```

Let's try it out! I'll register as a new user and then try to login. Once again, things work perfectly!

Chapter 22: Using an External Form Type Class

USING AN EXTERNAL FORM TYPE CLASS¶

We built our form right inside the controller, which was really simple. But, it makes our controller a bit ugly and if we needed to re-use this form somewhere else, that wouldn't be possible.

For those reasons, the code to create a form *usually* lives in its own class. Create a new `Form` directory and a new file called `RegisterFormType`. Create the class, give it a namespace and make it extend a class called `AbstractType`:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
namespace Yoda\UserBundle\Form;

use Symfony\Component\Form\AbstractType;

class RegisterFormType extends AbstractType
{
}
```

We need to add a few methods to this class. The first, and least important is `getName()`. Add this, and just return a string that's unique among your forms. It's used as part of the `name` attribute on your rendered form:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
// ...

public function getName()
{
    return 'user_register';
}
```

The really important method is `buildForm()`. I'm going to use my IDE to create this method for me. If you create your's manually, just don't forget the use statement for the `FormBuilderInterface`.

The `buildForm` method is where we build our form! Genius! Copy the code from our controller that adds the fields and put that here:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php

use Symfony\Component\Form\FormBuilderInterface;
// ...

public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('username', 'text')
        ->add('email', 'email')
        ->add('plainPassword', 'repeated', array(
            'type' => 'password'
        )
    );
}
```

Finally, create a `setDefaultOptions` function and set the `data_class` option on it:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php

use Symfony\Component\OptionsResolver\OptionsResolverInterface;
// ...

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Yoda\UserBundle\Entity\User',
    ));
}
```

That's it! This class is now a recipe for exactly how the form should look.

Using the Form Class¶

Remove the builder code in our controller. Instead, replace it with a call to `createForm` and pass it an instance of the new `RegisterFormType`:

```
// src/Yoda/UserBundle/Controller/RegisterController.php

use Yoda\UserBundle\Form\RegisterFormType;
// ...

public function registerAction(Request $request)
{
    $defaultUser = new User();
    $defaultUser->setUsername('Foo');

    $form = $this->createForm(new RegisterFormType(), $defaultUser);

    // ...
}
```

Refresh! We've conquered forms!

Forms: From Space¶

Some quick review. A form is something you build, giving it the fields and field types you need. At first, a form just returns an associative array, but we can change that with the `data_class` option to return a populated object. Forms can also be built right inside the controller or in an external class.

Got it? Great! Let's move on to validation.

Chapter 23: Registration Validation

REGISTRATION VALIDATION¶

Let's add some validation. What if I don't enter a valid email address or I choose a username that's already taken? Right now, the form would submit just fine, which is lame.

HTML5 Validation¶

Try to submit a blank form right now! Woh! We *do* have some validation. In fact, if I enter an invalid email address, we see another error. This is HTML5 validation. When I inspect a field, we see what's triggering it:

```
<input type="email" id="user_register_email" name="user_register[email]" required="required" />
```

First, each field has a `required` attribute, which tells an HTML5-compliant browser to throw an error if the field is left blank. Second, the input `type=email` field tells the browser to expect a valid email address instead of any string.

We got an input `type=email` field because we're using the `email` field type in our form. But where are these `required` attributes coming from?

Field Options¶

To answer that, look at the third argument when adding a field: the options array:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
// ...

$builder->add('email', 'email', array(
    // an array of options to pass to this field
))
```

Every field type can be configured in different ways. For example, the `repeated` field has a `type` option. There are also a bunch of options that every field has, `required` being one of them. Set the `required` option on email to false and refresh:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
// ...

$builder
    // ...
    ->add('email', 'email', array(
        'required' => false
    ))
    // ...
;
```

When we inspect, we see that the `required` attribute is gone. All fields have this option, and it's important to realize that it defaults to `true`.

There are a lot more options available for each field type. The easiest way to learn about them is via the documentation. Remember that Form Field Type Reference page we saw earlier? Yep, it shows you all the field types *and* all of their options.

Digging into the Core¶

You can also dig into the source code to find out what options are available. For the `repeated` type, there is a class called, well, `RepeatedType`. The `setDefaultOptions` method shows you the options that are special to this type.

Most of the global options are inherited from a class called `FormType`. In here you can see `required` and a few others. In its parent class, `BaseType`, we see a few more, like `label` and `attr`. We can use these to customize the label or add a class to the field from right inside the form class:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
// ...

$builder
    ->add('email', 'email', array(
        'required' => false,
        'label' => 'Email Address',
        'attr'    => array('class' => 'C-3PO')
    ))
    // ...
;
```

When we refresh, we see these options working for us.

Disabling HTML5 Validation¶

HTML5 validation is nice, but not enough: we still need server-side validation. Also, you can't really customize how HTML5 validation looks or its messages easily. And finally, Symfony automatically defaults all fields to have the `required` attribute, which is kind of annoying.

I recommend avoiding HTML5 validation entirely. To disable it, just add a `novalidate` attribute to your `form` tag:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}

{# ... #}
<form action="{{ path('user_register') }}" method="POST" novalidate="novalidate">
```

Refresh the form and try to submit empty. We get a *huge* error from the database which proves that HTML5 validation is off! Now let's add some server-side validation!

Chapter 24: Server-Side Validation

SERVER-SIDE VALIDATION¶

In Symfony, validation is done a little bit differently. Instead of validating the submitted form data itself, validation is applied to the `User` object.

Start by heading to the validation chapter of the documentation. Click on the “Annotations” tab of the code example and copy the `use` statement. Paste this into your `User` class:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

use Symfony\Component\Validator\Constraints as Assert;
```

Whenever you add annotations, you need a `use` statement.

Basic Constraints and Options¶

Adding a validation constraint is easy. To make the `username` field required, add `@Assert\NotBlank` above the property:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

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

Try it out! When we submit the form blank, we see the validation error above the field. It looks terrible, but we’ll work on that later. To customize the message, add the `message` option:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

/**
 * @ORM\Column(name="username", type="string", length=255)
 * @Assert\NotBlank(message="Put in a username you rebel scum :P")
 */
private $username;
```

Refresh to see the new error.

All of this magic happens automatically when we call `handleRequest` in our controller. This takes the submitted values, pushes them into the User object, and then applies validation.

Add all the Constraints!¶

Let’s keep going. We can use the `Length` constraint to make sure the `username` is at least 3 characters long:

```
/**
 * @ORM\Column(name="username", type="string", length=255)
 * @Assert\NotBlank(message="Put in a username of course!")
 * @Assert\Length(min=3, minMessage="Give us at least 3 characters!")
 */
private $username;
```

For the `email` property, use `NotBlank` *and* `Email` to guarantee that it's a valid email address:

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\NotBlank
 * @Assert\Email
 */
private $email;
```

For `plainPassword`, we can use the `NotBlank` constraint and the `Regex` constraint to guarantee a strong password:

```
/**
 * @Assert\NotBlank
 * @Assert\Regex(
 *     pattern="/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?!.*\s).*$/",
 *     message="Use 1 upper case letter, 1 lower case letter, and 1 number"
 * )
 */
private $plainPassword;
```

Let's try this out by filling out the form in different ways. All the errors show up! They're just really ugly.

Docs for The Built-In Constraints¶

Symfony comes packed with a lot of other constraints you can use. Check them out in the reference section of the documentation. You can see the `Length` constraint we just used and all of the options for it. Cool!

The UniqueEntity Constraint¶

Check out the UniqueEntity constraint. This is useful if you need to make sure a value stays unique in the database. We need to make sure that nobody signs up using an existing username or email address, so this is perfect.

The `UniqueEntity` constraint is special because unlike the others, this one requires a different `use` statement. Copy it into your `User` class. Also, `@UniqueEntity` goes *above* the class, not above a property. It takes two options: the field that should be unique followed by a message. Add a constraint for both the username and the email:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Table(name="yoda_user")
 * @ORM\Entity(repositoryClass="Yoda\UserBundle\Entity\UserRepository")
 * @UniqueEntity(fields="username", message="That username is taken!")
 * @UniqueEntity(fields="email", message="That email is taken!")
 */
class User implements AdvancedUserInterface, Serializable
```

Tip

`"username"` is equivalent to `fields="username"`. `fields` is the "default" option. If it's the only option you're using, saying `fields` isn't needed. See Constraint Configuration.

If we try to register with an existing username or email, we see the error!

The Callback Constraint¶

Before we move on, I want to show you one more useful constraint: Callback. This constraint is *awesome* because it lets you create a method inside your class that's called during validation. You can apply whatever logic you need to figure out if the object is valid. You can even place the errors on exactly which field you want. If you have a more difficult validation problem, this might be exactly what you need.

We won't show it here, but check it out.

Tip

You can also check Custom Validation, Callback and Constraints page of our "Question and Answer Day" tutorial to learn more about custom validation in Symfony with a bunch of examples and use cases.

Chapter 25: Adding a Flash Message

ADDING A FLASH MESSAGE¶

After registration, let's make the new user feel loved by giving them a big happy success message! Symfony has a feature called flash messages, which is perfect for this. A flash is a message that we set to the session, but that disappears after we access it exactly one time.

After registration, grab the `session` object from the request and get an object called a "flash bag". Set a message on it using `add`:

```
// src/Yoda/UserBundle/Entity/Controller/RegisterController.php
// ...

if ($form->isValid()) {
    // .. code that saves the user

    $request->getSession()
        ->getFlashBag()
        ->add('success', 'Welcome to the Death Star, have a magical day!')
    ;

    $url = $this->generateUrl('event');

    return $this->redirect($url);
}
```

Rendering a Flash Message¶

That's it! Now, if a flash message exists, we just need to print it on the page. Let's do that in `base.html.twig`. The session object is available via `app.session`. Use it to check to see if we have any `success` flash messages. If we do, let's print the messages inside a styled container. You'll typically only have one message at a time, but the flash bag is flexible enough to store any number:

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

<body>
    {% if app.session.flashBag.has('success') %}
        <div class="alert alert-success">
            {% for msg in app.session.flashBag.get('success') %}
                {{ msg }}
            {% endfor %}
        </div>
    {% endif %}

    <!-- ... -->
```

The `success` key is just something I made up. With this setup, whenever we need to show a happy message, we just need to set a `success` flash message and it'll show up here!

Let's test it out. We register, the flash message is set, and then it's displayed after the redirect. Nice!

Chapter 26: Automatically Authenticating after Registration

AUTOMATICALLY AUTHENTICATING AFTER REGISTRATION¶

After registration, let's log the user in automatically. Create a private function called `authenticateUser` inside `RegisterController`. Normally, authentication happens automatically, but we can also trigger it manually:

```
// src/Yoda/UserBundle/Entity/Controller/RegisterController.php
// ...
use Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken;

private function authenticateUser(User $user)
{
    $providerKey = 'secured_area'; // your firewall name
    $token = new UsernamePasswordToken($user, null, $providerKey, $user->getRoles());

    $this->container->get('security.context')->setToken($token);
}
```

This code might look strange, and I don't want you to worry about it too much. The basic idea is that we create a token, which holds details about the user, and then pass this into Symfony's security system.

Call this method right after registration:

```
// src/Yoda/UserBundle/Entity/Controller/RegisterController.php
// ...

if ($form->isValid()) {
    // .. code that saves the user, sets the flash message

    $this->authenticateUser($user);

    $url = $this->generateUrl('event');

    return $this->redirect($url);
}
```

Try it out! After registration, we're redirected back to the homepage. But if you check the web debug toolbar, you'll see that we're also authenticated as Padmé. Sweet!

Redirecting back to the original URL

Suppose an anonymous user tries to access a page and is redirected to `/login`. Then, they register for a new account. After registration, wouldn't it be nice to redirect them back to the page they were originally trying to access?

Yes! And that's possible because Symfony stores the original, protected URL in the session:

```
$key = '_security.'.$providerKey.'.target_path';
$session = $this->getRequest()->getSession();

// get the URL to the last page, or fallback to the homepage
if ($session->has($key)) {
    $url = $session->get($key)
    $session->remove($key);
} else {
    $url = $this->generateUrl('homepage');
}
```

The session storage key used here is pretty internal, and could change in the future. So use with caution!

Chapter 27: Functional Testing

FUNCTIONAL TESTING¶

Our site is looking cool. But how can we be sure that we haven't broken anything along the way? Right now, we can't!

Let's avoid the future angry phone calls from clients by adding some tests. There are two main types: unit tests and functional tests. Unit tests test individual PHP classes. We'll save that topic for another screencast. Functional tests are more like a browser that surfs to pages on your site, fills out forms and checks for specific things.

Your First Functional Test¶

When we generated the `EventBundle` in the last screencast, it created 2 stub functional tests for us. How nice!

Create a `Tests/Controller` directory in `UserBundle`, copy one of the test files and rename it to `RegisterControllerTest`:

```
// src/Yoda/UserBundle/Tests/Controller/RegisterControllerTest.php
namespace Yoda\UserBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class RegisterControllerTest extends WebTestCase
{
    public function testRegister()
    {
        $client = static::createClient();
        // ...
    }
}
```

Rename the method to `testRegister`:

```
// src/Yoda/UserBundle/Tests/Controller/RegisterControllerTest.php
// ...

public function testRegister()
{
    $client = static::createClient();
    // ...
}
```

The idea is that each controller, like `RegisterController` will have its own test class, like `RegisterControllerTest`. Then, each action method, like `registerAction`, will have its own test method, like `testRegister`. There's no technical reason you need to organize things like this. The only rule is that you need to start each method with the word "test".

Using the Client object¶

That `$client` variable is like a browser that we can use to surf to pages on our site. Start small by testing that the `/register` page returns a 200 status code and that the word "Register" appears somewhere:

```
public function testRegister()
{
    $client = static::createClient();

    $client->request('GET', '/register');
    $response = $client->getResponse();

    $this->assertEquals(200, $response->getStatusCode());
    $this->assertContains('Register', $response->getContent());
}
```

The `assertEquals` and `assertContains` methods come from PHPUnit, the library that will actually run the test.

Installing PHPUnit¶

To run the test, we need PHPUnit: the de-facto tool for testing. You can install it globally or locally in this project via Composer. For the global option, check out their docs.

Let's use Composer's `require` command and search for phpunit:

```
php composer.phar require
```

Choose the `phpunit/phpunit` result. For a version, I'll go to packagist.org and find the library. Right now, it looks like the latest version is `4.1.3`. I'll use the constraint `~4.1`, which basically means 4.1 or higher.

Tip

Want to know more about the ~ version constraint? Read Next Significant Release on Composer's website.

This added `phpunit/phpunit` to the `require` key in `composer.json` and it ran the `update` command in the background to download it.

Tip

Since PHPUnit isn't actually needed to make our site work (it's only needed to run the tests), it would be even better to put it in the `require-dev` key of `composer.json`. Search for `require-dev` on this post for more details.

Running the Tests¶

We now have a `bin/phpunit` executable, so let's use it! Pass it a `-c app` option:

```
php bin/phpunit -c app
```

Tip

If you're on Windows (or a VM running in Windows), the above command won't work for you (it'll just spit out some text). Instead, run:

```
bin\phpunit -c app
```

This tells PHPUnit to look for a configuration file in the `app/` directory. And hey! There's a `phpunit.xml.dist` file there already for it to read. This tells phpunit how to bootstrap and where to find our tests.

But we see a few errors. If you look closely, you'll see that it's executing the two test files that were generated automatically in `EventBundle`. Git rid of these troublemakers and try again:

```
rm src/Yoda/EventBundle/Tests/Controller/*Test.php
php bin/phpunit -c app
```

Green! PHPUnit runs our test, where we make a request to `/register` and check the status code and look for the word "Register".

To see what a failed test looks like, change the test to check for Ackbar instead of Resgister and re-run it:

```
$this->assertContains('Ackbar', $response->getContent());
```

It doesn't find it, but it does print out the page's content, which we could use to debug. It's a trap! Change the test back to look for `Register`:

```
$this->assertContains('Register', $response->getContent());
```

Traversing the Dom with the Crawler¶

When we call the `request()` function, it returns a `Crawler` object, which works a lot like the jQuery object in JavaScript. For example, to find the value of the username field, we can search by its `id` and use the `attr` function. It should be equal to "Leia":

```
public function testRegister()
{
    $client = static::createClient();

    $crawler = $client->request('GET', '/register');
    $response = $client->getResponse();

    $this->assertEquals(200, $response->getStatusCode());
    $this->assertContains('Register', $response->getContent());

    $usernameVal = $crawler
        ->filter('#user_register_username')
        ->attr('value')
    ;
    $this->assertEquals('Leia', $usernameVal);
}
```

Re-run the test to see the result:

```
php bin/phpunit -c app
```

Tip

To see everything about the crawler, check out The DomCrawler Component.

Chapter 28: Testing Forms

TESTING FORMS¶

One of the most common things you'll want to do in a test is fill out a form. Start by using the crawler to select our submit button and get a `Form` object. Notice that we're selecting it by the actual text that shows up in the button, though you can also use the `id` or `name` attributes:

```
public function testRegister()
{
    // ...

    // the name of our button is "Register!"
    $form = $crawler->selectButton('Register!')->form();
}
```

In the browser, if we submit the form blank, we should see the form again with some errors. We can simulate this by calling `submit()` on the client and passing it the `$form` variable.

Tip

Both `request()` and `submit()` return a Crawler object that represents the DOM after making that request. Be sure to always get a new `$crawler` variable each time you call one of these methods.

Let's test that the status code of the error page is 200 and that we at least see an error:

```
public function testRegister()
{
    // ...

    // the name of our button is "Register!"
    $form = $crawler->selectButton('Register!')->form();

    $crawler = $client->submit($form);
    $this->assertEquals(200, $client->getResponse()->getStatusCode());
    $this->assertRegexp(
        '/This value should not be blank/',
        $client->getResponse()->getContent()
    );
}
```

Run the test again!

```
php bin/phpunit -c app
```

Beautiful!

Filling out the Form with Data¶

Let's submit the form again, but this time with some data! Use `selectButton` to get another `$form` object.

Now, give each field some data. This is done by treating the form like an array and putting data in each field. These names come right from the HTML source code, so check there to see what they look like:

```
public function testRegister()
{
    // ...

    // submit the form again
    $form = $crawler->selectButton('Register!')->form();

    $form['user_register[username]'] = 'user5';
    $form['user_register[email]'] = 'user5@user.com';
    $form['user_register[plainPassword][first]'] = 'P3ssword';
    $form['user_register[plainPassword][second]'] = 'P3ssword';

    $crawler = $client->submit($form);
}
```

Now when we submit, the response we get back should be a redirect. We can check that by calling the `isRedirect` method on the response. Next, use the `followRedirect()` method to tell the client to follow the redirect like a standard browser. Finally, let's make sure that our success flash message shows up after the redirect:

```
public function testRegister()
{
    // ...

    $crawler = $client->submit($form);
    $this->assertTrue($client->getResponse()->isRedirect());
    $client->followRedirect();
    $this->assertContains(
        'Welcome to the Death Star, have a magical day!',
        $client->getResponse()->getContent()
    );
}
```

Run the tests!

```
php bin/phpunit -c app
```

Success! We now have proof that we can visit the registration form and fill it out with and without errors. If we accidentally break that later, our test will tell us.

Chapter 29: Controlling Data / Fixtures in a Test

CONTROLLING DATA / FIXTURES IN A TEST¶

When we try running the test again, it fails! What the heck?

If we did a little debugging we'd see that it fails the second time because the username and email in the test are already taken. So instead of success, we see a validation error.

This is a *very* important detail about testing. Before running any test, we need to make sure the database is in a predictable state. Our test will pass... unless there *happens* to already be a `user5` in the database.

Deleting Users¶

To fix this, let's empty the user table before running the test. Start by grabbing the container object, which is stored statically on the parent test class:

```
// src/Yoda/UserBundle/Tests/Controller/RegisterControllerTest.php
// ...

public function testRegister()
{
    $client = static::createClient();

    $container = self::$kernel->getContainer();
    // ...
}
```

Now, grab the Doctrine entity manager by getting the `doctrine` service and calling `getManager`. If you're not comfortable with what I just did, don't worry. I'll demystify this in a moment:

```
// src/Yoda/UserBundle/Tests/Controller/RegisterControllerTest.php
// ...

public function testRegister()
{
    $client = static::createClient();

    $container = self::$kernel->getContainer();
    $em = $container->get('doctrine')->getManager();

    // ...
}
```

Once we have the entity manager we can get the `UserRepository`. Build a query from the repository that deletes all of the users:

```
// src/Yoda/UserBundle/Tests/Controller/RegisterControllerTest.php
// ...

public function testRegister()
{
    // ...
    $em = $container->get('doctrine')->getManager();
    $userRepo = $em->getRepository('UserBundle:User');
    $userRepo->createQueryBuilder('u')
        ->delete()
        ->getQuery()
        ->execute()
    ;

    // ... the actual test
}
```

Re-run the test again to see that everything passes. Now this is a good-looking test!

Notice that we're not testing for every tiny detail. That would be a ton of work and functional tests aren't meant to replace us as developers from making sure things work when we build them.

Instead, we just want to see that the form can be submitted successfully and unsuccessfully. The important thing is that if we break something major later, our test will let us know.

Separating the dev and test Databases¶

When we run our tests, it's emptying the same user table we're using for development! That's kind of annoying! Instead, let's use 2 different databases: one for development and one that's used only by the tests.

Open up the main `config.yml` file and find the doctrine database configuration. Copy and paste it into the `config_test.yml` file, but remove everything except for the `dbname` option. Now, add an `_test` to the end of it:

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

doctrine:
    dbal:
        dbname:   %database_name%_test
```

Tip

The `database_name` is a parameter, which lives in `app/config/parameters.yml`.

This little trick takes advantage of how Symfony environments work. The `test` environment uses all of the normal Doctrine configuration, except that it overrides this one option.

Don't forget to setup the new database by running `doctrine:database:create`. Pass an extra `--env=test` option so that the command runs in the `test` environment. Use the same idea to insert the schema:

```
php app/console doctrine:database:create --env=test
php app/console doctrine:schema:create --env=test
```

Tip

By default, all `app/console` commands run in the `dev` environment.

You can now re-run the tests knowing that our main database isn't being affected:

```
php bin/phpunit -c app
```

Behat¶

As cool as this is, in reality we use a tool called Behat instead of Symfony's built in functional testing tools. And you're in luck because everything you just learned translates to Behat. Check out our tutorial on this to take your functional testing into space!

Do this or risk an angry phone call from Darth Vader when the super laser doesn't fire because you added a new espresso machine to the breakroom.

Chapter 30: More about Container, the “doctrine” Service and the Entity Manager

MORE ABOUT CONTAINER, THE “DOCTRINE” SERVICE AND THE ENTITY MANAGER¶

In our test, we needed Doctrine’s entity manager and to get it, we used Symfony’s container. Remember from the first episode in this series that the “container” is basically just a big array filled with useful objects. To get a list of all of the objects in the container, run the `container:debug` console command:

```
php app/console container:debug
```

One of these objects is called `doctrine`, which is an instance of a class called `Registry`:

```
...
doctrine    container    Doctrine\Bundle\DoctrineBundle\Registry
```

So when we say `$container->get('doctrine')`, we’re getting this object.

Find the shortcut in your editor that can open files by typing in their filename. Use this to find and open `Registry.php`. Inside, you’ll see the `getManager` method being used, which actually lives on its parent class.

The Base Controller¶

So how does this compare with how we normally get the entity manager in a controller? Open up `RegisterController`. That’s right - in a controller, we always say `$this->getDoctrine()->getManager()`:

```
$em = $this->getDoctrine()->getManager();
```

The `getDoctrine()` method lives inside Symfony’s `Controller` class, which we’re extending. Let’s open up that class to see what this method does:

```
// vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php
public function getDoctrine()
{
    return $this->container->get('doctrine');
}
```

Ah-hah! The `getDoctrine` method is just a shortcut to get the service object called `doctrine` from the container. This means that no matter where we are, the process to get the entity manager is always the same: get the container, get the `doctrine` service, then call `getManager` on it.

My big point is that if you have the container, then you can get *any* object in order to do *any* work you need to. The only tricky part is knowing what the name of the service object is and what methods you can call on it. Using the `app/console container:debug` task can help.

Chapter 31: After-dinner Mint

AFTER-DINNER MINT¶

The site is looking sweet! And now with most of the work behind us, let's relax a little and have some fun. In this last part, we'll check out some cool things related to forms and security.

Form Field Guessing¶

Remember when we disabled HTML5 validation earlier. Let's add it back temporarily. Remove the `novalidate` attribute so that it works again:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}

{# ... #}
<form action="{{ path('user_register') }}" method="POST">
```

Now, open the `User` class: let's do a little experimenting. Let's pretend that the `email` field isn't requied. Remove the `NotBlank` constraint from it and set a `nullable=true` option in the Doctrine metadata. Don't worry about updating your schema - this change is just temporary:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

/**
 * @ORM\Column(type="string", length=255, nullable=true)
 * @Assert\Email
 */
private $email;
```

Now, open up `RegisterFormType` and remove the `required` option, which determines whether or not the HTML5 `required` attribute should be rendered:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
// ...

$builder
    // ...
    ->add('email', 'email', array(
        'label' => 'Email Address',
        'attr'    => array('class' => 'C-3PO')
    ))
    // ...
;
```

When we surf to the registration page and try to submit, HTML5 validation stops us. And just like before, the `email` field has the `required` attribute on it. We saw earlier that we can fix the problem by setting the `required` option to `false` for that field. But shouldn't the form be able to see that the `email` field isn't required in `User` and set the option to `false` for us?

Actually, it can! The feature is called "field guessing" and it works like this. Set the second argument of `add` for the `email` field to null:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
// ...

$builder
    // ...
    ->add('email', null, array(
        'required' => false,
        'label' => 'Email Address',
        'attr'    => array('class' => 'C-3PO')
    ))
    // ...
;
```

This might seem a little crazy, because this argument normally tells Symfony what type of field this is. Will it be able to figure how to render this field?

Refresh the page and inspect `email` - there are a bunch of awesome things happening:

```
<input type="email" id="user_register_email" name="user_register[email]" maxlength="255" />
```

First, notice that the field is still `type="email"`. That's being guessed based on the fact that there is an `Email` constraint on the property. Remove the `Email` constraint and refresh:

```
// src/Yoda/UserBundle/Entity/User.php
// ...

/**
 * @ORM\Column(type="string", length=255, nullable=true)
 */
private $email;
```

```
<input type="text" id="user_register_email" name="user_register[email]" maxlength="255" />
```

Symfony doesn't know anything about the field now, so it just defaults to the `text` type.

Field Option Guessing¶

But now, notice that the `required` attribute is gone. In addition to guessing the field type, certain options are also guessed, like the `required` option. Let's play with this. Add back the `NotBlank` constraint and refresh:

```
/**
 * @ORM\Column(type="string", length=255, nullable=true)
 * @Assert\NotBlank()
 */
private $email;
```

Not surprisingly, the `required` attribute is back. Next, remove `NotBlank`, but also make the field `not null`:

```
/**
 * @ORM\Column(type="string", length=255, nullable=false)
 */
private $email;
```

Yep, the `required` attribute is *still* there. The form system guesses that the field is required based on the fact that it's required in the database.

Even the `maxlength` attribute that's being rendered comes from the length of the field in the database.

So here's the deal. If you leave the second argument empty when creating a field, Symfony will try to guess the field type *and* some options, like `required`, `max_length` and `pattern`. Field guessing isn't always perfect, but I tend to try it at first and explicitly set things that aren't guessed correctly.

Let's put our 2 validation constraints back and add back the `email` type option in the form and refresh:

```
// src/Yoda/UserBundle/Entity/User.php
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\NotBlank
 * @Assert\Email
 */
private $email;
```

```
// src/Yoda/UserBundle/Form/RegisterFormType.php

$builder
    // ...
    ->add('email', 'email')
    // ...
;
```

If you were watching closely, the `maxlength` attribute disappeared:

```
<input type="text" id="user_register_email" name="user_register[email]" required="required" />
```

This is a gotcha with guessing. As soon as you pass in the `type` argument, none of the options like `required` or `max_length` are guessed anymore. In other words, if you don't let Symfony guess the field type, it won't guess any of the options either.

Chapter 32: Security: Creating Roles and Role Hierarchies

SECURITY: CREATING ROLES AND ROLE HIERARCHIES¶

Let's change gears and mention a few more things about security. Earlier, we saw how you could enforce security in two different ways. The `access_control` method is the easiest, but we can always enforce things manually in the controller. In both cases, we're checking whether or not a user has a specific role. If they do, they get access. If they don't, they'll see the login page or the access denied screen.

In our example, we showed a pretty basic system with just `ROLE_USER` and `ROLE_ADMIN.` If you need another role, just start using it. For example, if only *some* users are able to create events, we can protect event creation with a new role.

To show this, let's make the role that's passed to `enforceUserSecurity` configurable and then only let a user create an event if they have some `ROLE_EVENT_CREATE` role:

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

public function createAction(Request $request)
{
    $this->enforceUserSecurity('ROLE_EVENT_CREATE');
}

// also change ROLE_USER to ROLE_EVENT_CREATE in newAction

// ...
private function enforceUserSecurity($role = 'ROLE_USER')
{
    $securityContext = $this->container->get('security.context');
    if (!$securityContext->isGranted($role)) {
        // in Symfony 2.5
        // throw $this->createAccessDeniedException('message!');
        throw new AccessDeniedException('Need '.$role);
    }
}
```

The *only* rule when creating a role is that it *must* start with `ROLE_`. If it doesn't, you won't get an error, but security won't be enforced.

Try it out by logging in as admin. But first, reload the fixtures, since our users were deleted earlier when running our functional test.

Now, try to create an event. No access! Our admin user has `ROLE_USER` and `ROLE_ADMIN`, but not `ROLE_EVENT_CREATE`. If we want to give all administrators the ability to create events, we can take advantage of role hierarchy, which we can see in `security.yml`. Add `ROLE_EVENT_CREATE` to `ROLE_ADMIN` and refresh again:

```
# app/config/security.yml
security:
    # ...
    role_hierarchy:
        ROLE_ADMIN:       [ROLE_USER, ROLE_EVENT_CREATE]
        ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

We are in! Now let's schedule that wookiee wine down!

STRATEGIES FOR CONTROLLER ACCESS¶

Keep these two tips in mind when using roles:

1. Protect the actual parts of your application using feature-specific roles, not user-specific roles. This means your roles should describe the features they give you access to, like `ROLE_EVENT_CREATE` and not the type of user that should have access, like `ROLE_ADMIN`.

2. Use the role hierarchy section to manage which types of users have which roles. For example, you might decide that `ROLE_USER` should have `ROLE_BLOG_CREATE` and `ROLE_EVENT_CREATE`, which you setup here. Assign your actual users these user-specific roles, like `ROLE_USER` or `ROLE_MARKETING`.

By following these tips, you'll be able to easily control the exact areas of your site that different users have access to.

Chapter 33: Switching Users / Impersonation

SWITCHING USERS / IMPERSONATION¶

What's that `ROLE_ALLOWED_TO_SWITCH` all about in `security.yml`. Symfony gives you the ability to actually *change* the user you're logged in as. Ever have a client complaint you couldn't replicate? Well now you can login as them without knowing their password. Now that is a Jedi mindtrick.

To activate this feature, add the `switch_user` key to your firewall:

```
# app/config/security.yml
security:
    # ...
    firewalls:
        secured_area:
            # ...
            switch_user: ~
```

To use it, just add a `?_switch_user=` query parameter to any page with the username you want to change to:

http://events.local/app_dev.php/new?_switch_user=darth

When we try it initially, we get the access denied screen. Our user needs `ROLE_ALLOWED_TO_SWITCH` to be able to do this. Add it to the `ROLE_ADMIN` hierarchy to get it:

```
# app/config/security.yml
security:
    # ...
    role_hierarchy:
        ROLE_ADMIN:     [ROLE_USER, ROLE_EVENT_CREATE, ROLE_ALLOWED_TO_SWITCH]
        # ...
```

When we refresh, you'll see that the our username in the web debug toolbar has changed to darth. So cool! To switch back, use the `_exit` key:

```
http://events.local/app_dev.php/new?_switch_user=_exit
```

Chapter 34: Whitelisting: Securing all Pages, except a few

WHITELISTING: SECURING ALL PAGES, EXCEPT A FEW¶

Next look again at `access_control`. Right now, our entire site is open to the public, except for the specific pages that we're locking down in our controller.

But what if almost *every* page on our site required login? Is there a nice way to enforce this?

Add a new access control that matches *all* requests and requires `ROLE_USER`:

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/, roles: ROLE_USER }
```

Now, every page is locked down. Logout and try it. Mmm a redirect loop!

We've got too much security. When we go to any page, we don't have access and are redirected to `/login`. Of course, we don't have access to `/login` either, so we're redirected to `/login`. Do you see the problem?

To fix this, add a new `access_control` entry *above* this for any page starting with `/login`. For the role, type `IS_AUTHENTICATED_ANONYMOUSLY`:

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/, roles: ROLE_USER }
```

Refresh again. It works! We're missing our styles, but we'll fix that next. The `access_control` entries match from top to bottom and stop after the *first* match. When we go to `/login`, the first control is matched and executed. By saying `IS_AUTHENTICATED_ANONYMOUSLY`, we're "whitelisting" this URL pattern because *every* user, even anonymous users, have this role.

Run the `router:debug` task to see a few other URLs that we should whitelist. These include some URLs that load our CSS files as well as the web debug toolbar and profiler during development. We also need to let anonymous users get to the registration page:

> _assetic_01e9169 ANY /css/01e9169.css ... _wdt ANY /_wdt/{token} _profiler_home ANY /_profiler/ user_register ANY /register ...

We haven't talked about assetic much yet, but by blocking it's URLs, we're blocking our stylesheets. With these entries in place, we're in good shape:

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/register, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/(css|js), roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/(_wdt|_profiler), roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/, roles: ROLE_USER }
```

Chapter 35: Accessing the User

ACCESSING THE USER¶

Now that we're logged in, how can we get access to the User object?

In a Template¶

Open up the homepage template. In Twig, we can access the User object by calling `app.user`. Let's use it to print out the username:

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

{# ... #}
{% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
    <a class="link" href="{{ path('logout') }}">
        Logout {{ app.user.username }}
    </a>
{% endif %}
```

Tip

If the user isn't logged in, `app.user` will be null. So be sure to check that the user is logged in first before using `app.user`.

Accessing the User in a Controller¶

From a controller, it's just as easy. Go to the controller function for the homepage and grab an object called the security context. Then call `getToken()` and `getUser()`:

```
public function indexAction()
{
    $user = $this->container
        ->get('security.context')
        ->getToken()
        ->getUser()
    ;
    var_dump($user->getUsername());die;
    // ...
}
```

Actually, since this is a bit long, the Symfony base controller gives us a shortcut method called `getUser`:

```
public function indexAction()
{
    $user = $this->getUser();
    var_dump($user->getUsername());die;
    // ...
}
```

I showed you the longer option first so that you'll understand that there is a service called `security.context` which is your key to getting the current `User` object. Remove this debug code before moving on.

Chapter 36: Remember Me Functionality

REMEMBER ME FUNCTIONALITY¶

I want to leave you with just one more tip. We talked a bit about the remember me functionality, but we didn't actually see how to use it. Activate the feature by adding the `remember_me` entry to your firewall and giving it a secret, random key:

```
# app/config/security.yml
security:
    # ...
    firewalls:
        secured_area:
            # ...
            remember_me:
                key: "Order 1138"
```

Tip

You can also use a *secret* parameter from *parameters.yml* as a remember me key to centralize secret key management for the entire application.

Next, open the login template and add a field named `_remember_me`:

```
{# src/Yoda/UserBundle/Resources/views/Login/login.html.twig #}
{# ... #}

<form ...>

    <hr/>
    Remember me <input type="checkbox" name="_remember_me" />
    <button type="submit" class="btn btn-primary pull-right">login</button>
</form>
```

This works a bit like login does: as long as we have a `_remember_me` checkbox and it's checked, Symfony will take care of everything automatically.

Try it out! After logging in, we now have a `REMEMBERME` cookie. Let's clear our session cookie to make sure it's working. When I refresh, my session is gone but I'm still logged in. Nice! Click anywhere on the web debug toolbar to get into the profiler. Next, click on the "Logs" tab. If you look closely, you can even see some logs for the remember me login process:

```
DEBUG - Remember-me cookie detected.
INFO - Remember-me cookie accepted.
DEBUG - SecurityContext populated with remember-me token.
```

Ok gang, that's all for now! I hope I'll see you in future Knp screencasts. And remember to check out KnpBundles.com if you're curious about all the open source bundles that you can bring into your app. Seeya next time!